JFP **14** (3): 263–315, May 2004. © 2004 Cambridge University Press
DOI: 10.1017/S0956796803004775 Printed in the United Kingdom

The recursive record semantics of objects revisited

GÉRARD BOUDOL
INRIA Sophia Antipolis, BP 93 – 06902 Sophia Antipolis Cedex, France

Abstract

In a call-by-value language, representing objects as recursive records requires using an unsafe fixpoint. We design, for a core language including extensible records, a type system which rules out unsafe recursion and still supports the construction of a principal type for each typable term. We illustrate the expressive power of this language with respect to object-oriented programming by introducing a sub-language for "mixin-based" programming.

1 Introduction

During the past fifteen years there has been very active research about the formalization of object-oriented programming concepts. One of the main purposes of this research was to design operational models of objects supporting rich type systems, so that one could benefit both from the flexibility of the object-oriented style, and from the safety properties guaranteed by typing. Let us be more precise here: our goal is to have an expressive language – as far as object-oriented constructs are concerned – with a type discipline *à la* ML (Damas & Milner, 1982; Milner, 1978), i.e. implicit typing with assignment of a principal type, ruling out run-time errors. This goal has proven difficult to achieve, and most of the many proposals that were put forward fall short of achieving it – with the exception of OCaml (Leroy *et al.*, 2000; Rémy & Vouillon, 1998), that we will discuss later.

While the meaning of "typing *à la* ML" should be clear, it is perhaps less easy to see what is meant by "object-oriented". We do not claim to answer to this question here. Let us just say that, in our view, objects encapsulate a state and react to messages, i.e. method invocations, by updating their state and sending messages, possibly to themselves. Moreover, in our view, object-orientation also involves inheritance, which includes – but should not be limited to, as we shall see – the ability to add and redefine methods. With this informal notion of object-orientation in mind, let us review some of the proposals we alluded to.

An elegant proposal was made by Wand (1994), based on his *row variables* (Wand, 1987), consisting of a class-based model, where classes are functions from instance variables and a "self" parameter to extensible records of methods, and objects are fixpoints of instantiated classes, that is, recursive records. In this model invoking the method of an object amounts to selecting the corresponding component of the record representing the object. An operation of record extension is used to provide

a simple model of inheritance, *à la* SMALLTALK: a class B inherits from a class A by adding to it new methods, or redefining (overriding) some of them. Unfortunately, although its elegance and simplicity make it very appealing, Wand's model is not expressive enough. More specifically, it does not support state changes in objects: one may override a method in an inherited class, but one apparently cannot modify the state of the object during its life-time (see, for instance, Abadi & Cardelli 1996, Section 6.7.2). This is because in creating the object, the "self" parameter is bound too early.

Wand's model is an instance of what is known as the *recursive record semantics* for objects (see Fischer & Mitchell, 1995), initiated by Cardelli (1984). Based on this idea that an object is the fixpoint of an instantiated class, Cook proposed a more elaborate model (Cook *et al.*, 1994), where updating the state of an object is possible, by creating new objects, instances of the same class. Then a class is also recursive in Cook's model, since methods may have to call a "myClass" parameter. This model is operationally quite expressive, but the type theory that it uses is also quite elaborate, and does not fulfil our desires, of existence of an algorithm for assigning a (principal) type to each typable expression. The same remark actually applies to all the object models that use higher-order types (Abadi & Cardelli, 1996; Bruce, 1993; Eifrig *et al.*, 1995b; Fisher *et al.*, 1993; Pierce & Turner, 1994).

In another approach, due to Kamin (1988) and known as the *self-application semantics*, an object is a record of pre-methods, that are functions of the object itself. The object is bound to self only when a method is invoked, by applying the pre-method to the object. In this way, the state of the object may dynamically be updated. In this model, which looks indeed operationally satisfactory, an object is not quite a record, since from a typing point of view, we must know that the first parameter (that is, self) of all its pre-methods have the same type. In other words, one must have in this approach specific constructs for objects and object types, depending on the type of self, thus different from record types. This has been developed in *object calculi*, most notably by Fisher and Mitchell (Fisher, 1996; Fisher *et al.*, 1993; Fisher & Mitchell, 1995) – who call it "the axiomatic approach" – and Abadi and Cardelli (Abadi, 1994; Abadi & Cardelli, 1996), but as we already noticed, in calculi that support a rich form of inheritance, and in particular object extension, like Fischer *et al.* (1993), the type theory which is used is quite elaborate, and does not support implicit typing.

Object calculi claim to fix the principles for objects, thus providing simple formal models, but they actually take design decisions, about inheritance in particular – a concept which is still a matter of debate in the object-oriented programming community (see Taivalsaari (1996), for example). As a matter of fact, many of the proposals for an object model, including OCAML, follow this approach of designing a specific calculus (Abadi & Cardelli, 1996; Bono *et al.*, 1999a; Bruce, 1993; Fisher *et al.*, 1993; Rémy & Vouillon, 1998). However, there could be some benefits from deriving object-oriented concepts from more basic principles: first, their typing could be derived within simple, unquestionable typing systems (see, for instance, the comments in McQueen (2002)). Secondly, they could be better integrated in a standard computational model, in which one could formalize and compare various

approaches to objects, and get more flexible object models. Furthermore, we would not have to develop specific theories for reasoning about them.

In this paper, we pursue Wand's approach, aiming at encoding object-oriented concepts by means of extensible records. One may observe that the update operation of object calculi (Abadi & Cardelli, 1996; Fisher *et al.*, 1993) is actually overloaded: it serves both in inheritance, to override methods, and in the dynamic behaviour of an object, to update the state (see Abadi & Cardelli, 1996, Section 5.2). As we have seen, the first usage is not problematic in Wand's model, whereas the second is. Then an obvious idea is to abandon the "functional update" approach in favor of a rather more natural *imperative update* approach (Abadi & Cardelli, 1996; Bono *et al.*, 1999a; Eifrig *et al.*, 1995b; Rémy & Vouillon, 1998). This means that we are in a language with *references* (following ML's terminology), where a call-by-value strategy is assumed for evaluation. Now a new problem arises: to build objects as recursive records one must have the ability to build recursive non-functional values, and this, in principle, is not supported in a typed call-by-value language. More specifically, we would like to use the construct (let rec $x = N$ in M), where N may be of a record type. This is evaluated by first computing a value for N, returning a cyclic binding to this value for x, and then computing M. Notice that side effects and creation of new references arising from the evaluation of N are completed before a cyclic binding is returned. This is what we need to install the state of an object before returning its (recursive) record of methods. The resulting object model is similar to what is known as the "cyclic record" encoding, see Abadi & Cardelli, 1996, Sections 18.2.4 and 18.3.4 (see also Eifrig *et al.* (1995a), which includes a discussion about how to compensate for the lack of a general fixpoint).

As remarked by Rémy (1994a), a recursive record semantics of objects works fine with the let rec construct, except that this construct is *unsafe*. Indeed, some langages, like SCHEME or OCAML, provide us with this feature, but, except for defining recursive functions, its semantics is implementation-dependent. More precisely, in computing (let rec $x = N$ in M), it could happen that evaluating N we have to call the value of x, which is not yet computed, thus getting stuck at this point. An example, in the simply-typed call-by-value λ-calculus with recursion, is (let rec $x = \mathsf{F}(xV)$ in $\cdots$) where F is the combinator $\lambda x \lambda y\, y$ and V is any typable value. One must then have means to prevent such a run-time error in order to design a "safe" object model from recursive records. We must point out that, although this problem of determining restrictions on recursive definitions to ensure that they define something is not at all a new one, no obvious solution to our specific problem emerges from the literature, since we have to sometimes accept (let rec $x = (Gx)$ in M), especially when G reduces to a "generator" $\lambda \mathsf{self} M$ (Cook & Palsberg, 1989).

The main contribution of this paper is a solution to this problem: first, we extend the core "Reference ML" language, as considered by Wright & Felleisen (1994), with let rec and operations on records, similar to the ones of Cardelli & Mitchell (1994). We then provide a type system for this language, refining the simple typing by assigning a boolean information, 1 or 0 – that we call the "safeness degree" –, to variables in the typing context. This "degree" is to be interpreted as "certainly safe", or conversely "possibly unsafe", for recursion. Typically, a variable occurring within

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

$$
\begin{array}{rcll}
M, N \ldots & ::= & & \textit{expressions} \\
 & & V \mid (MN) \mid (\mathsf{let}\ D\ \mathsf{in}\ M) & \textit{core constructs} \\
 & \mid & \langle M, \ell = N \rangle \mid (M.\ell) \mid (M \backslash \ell) & \textit{record operations} \\
V, W \ldots & ::= & x \mid \mathsf{ref} \mid \mathbf{!} \mid \mathsf{set} \mid (\mathsf{set}\, V) & \textit{values} \\
 & \mid & \lambda x M \mid () \mid R & \\
R & ::= & x \mid \langle\rangle \mid \langle R, \ell = V \rangle & \textit{record values} \\
D & ::= & x = N \mid \mathsf{rec}\ x = N & \textit{declarations}
\end{array}
$$

Fig. 1. Syntax.

a value is safe for recursion, hence may have degree 1 – this is basically the standard approach, where recursion is "guarded", like for instance in (let rec $f = \lambda x N$ in M). We also have to introduce degrees in function types, considering types of the form $\theta^d \to \tau$. Then a function of type $\theta^1 \to \tau$ is "protective" towards its argument, like for instance $\mathsf{K} = \lambda x \lambda y\, x$, or more generally $\lambda x V$. For a recursion (let rec $x = (Gx)$ in M) to be safe, the function G must be "protective". Regarding our type system, we prove the standard properties: we show that the evaluation of a typable term either diverges or returns a value, thus avoiding to get stuck in run-time errors, and, adapting a result by Jategaonkar & Mitchell (1993), we show that a principal type may be computed for any typable expression. To assess the usefulness of the approach, and to illustrate the expressive power of the model, we introduce a few derived constructs for a *mixin-based* style of programming.

The rest of the paper is organized as follows: a first section introduces the language, from an operational point of view. We characterize in particular the possible outcomes of a computation. Section 3 introduces the type system. In the next one the type safety result is established. This relies upon the fact that substituting a term of appropriate type for a variable preserves the typing; the proof of this fact is more difficult than usual, because we have to deal with degrees. In Section 5 we present the type assignment algorithm, and prove the principal type property. In Section 6 we introduce our sub-language for mixin-based programming, and its derived typing. We illustrate the flexibility of the approach by means of a series of examples. Finally, we discuss related work, and present some conclusions.

Note. The type system presented in this paper is similar to, but not the same as, that presented in the conference version of this work (Boudol, 2001), which appeared to be not expressive enough for the purpose of typing the object-oriented constructs presented below. This is explained in Section 6.

2 The calculus

Assuming that a set $\mathscr{X}$ of variables, ranged over by x, y, $z \ldots$, and a set $\mathscr{L}$ of labels are given, the syntax of our core language is given in figure 1, where $x \in \mathscr{X}$ and $\ell \in \mathscr{L}$. It contains the "Reference ML" calculus of Wright & Felleisen (1994) – defining the call-by-value fixpoint combinator Y as (let rec $y = \lambda f.f\ \lambda x.yfx$ in y), where we use

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

$$\begin{array}{rcl} \mathbf{E} & ::= & [] \mid (\mathbf{E}N) \mid (V\mathbf{E}) \mid (\mathsf{let}\ x = \mathbf{E}\ \mathsf{in}\ M) \mid (\mathsf{let\ rec}\ x = \mathbf{E}\ \mathsf{in}\ M) \\ & \mid & \langle \mathbf{E}, \ell = N\rangle \mid \langle R, \ell = \mathbf{E}\rangle \mid (\mathbf{E}.\ell) \mid (\mathbf{E}\backslash\ell) \end{array}$$

Fig. 2. Evaluation contexts.

$$\begin{array}{rcll} (\lambda x M V) & \to & \{x \mapsto V\}M & \\ (\mathsf{let}\ x = V\ \mathsf{in}\ M) & \to & \{x \mapsto V\}M & \\ (\mathsf{let\ rec}\ x = V\ \mathsf{in}\ M) & \to & \{x \mapsto (\mathsf{let\ rec}\ x = V\ \mathsf{in}\ V)\}M & \\ (\langle R, \ell = V\rangle.\ell) & \to & V & \\ (\langle R, \ell = V\rangle.\ell') & \to & (R.\ell') & \ell' \neq \ell \\ (\langle R, \ell = V\rangle\backslash\ell) & \to & R & \\ (\langle R, \ell = V\rangle\backslash\ell') & \to & \langle (R\backslash\ell'), \ell = V\rangle & \ell' \neq \ell \\ M \to M' & \Rightarrow & \mathbf{E}[M] \to \mathbf{E}[M'] & \end{array}$$

Fig. 3. Local reduction.

the standard abbreviations, namely $\lambda x_1 \ldots x_n.M$ for $\lambda x_1 \ldots \lambda x_n M$ and $MN_1 \cdots N_k$ for $(\cdots(MN_1)\cdots N_k)$, and denoting := by set. Free (fv) and bound (bv) variables are defined as usual, and we denote by $\{x \mapsto N\}M$ the capture-free substitution. As usual we write $(M\ ;\ N)$ for $(\mathsf{let}\ x = M\ \mathsf{in}\ N)$ provided that x does not occur in N.

Regarding records, we use the operations of Cardelli & Mitchell (1994), denoting by $\langle M, \ell = N\rangle$ the record M extended with a new field, labelled ℓ, with value N. As in Cardelli & Mitchell (1994), this will only be well-typed if M does not exhibit an ℓ field, whereas the restriction operation, still denoted $(M\backslash\ell)$ and consisting of removing the ℓ field from M, will only be well-typed here if M does contain an ℓ field. This record calculus is equivalent to the one defined by means of pattern matching in Jategaonkar & Mitchell (1993): using abstraction on patterns, one may define selection and restriction respectively as $\lambda\langle x, \ell = y\rangle y$ and $\lambda\langle x, \ell = y\rangle x$. Conversely, the expression $\lambda\langle x, \ell = y\rangle M$ for instance may be written $\lambda z(\mathsf{let}\ x = z\backslash\ell\ \mathsf{in}\ (\mathsf{let}\ y = z.\ell\ \mathsf{in}\ M))$. The overriding operation is denoted $\langle M, \ell \leftarrow N\rangle$; this is an abbreviation for $\langle (M\backslash\ell), \ell = N\rangle$. We may also define the renaming operation $M[\ell \leftarrow \ell'] = (\mathsf{let}\ x = M\ \mathsf{in}\ \langle x\backslash\ell, \ell' = x.\ell\rangle)$. We shall write $\langle \ell_1 = M_1, \ldots, \ell_n = M_n\rangle$ for the record $\langle \cdots \langle \langle\rangle, \ell_1 = M_1\rangle \ldots, \ell_n = M_n\rangle$.

Now we specify the semantics of our language, defining first an *evaluation relation* $M \to M'$, that we also call *local* (or functional) *reduction*. The axioms and rules are given in figure 3. Here, as it is standard, we use a semantics by substitution for the let construct, and for β_v-reduction. This will simplify the proofs of the technical results, since we do not enter into the details of α-conversion, which is a standard matter. Obviously, an implementation would rather be closer to an abstract machine description, based on environments and closures (see Boudol & Zimmer (2002)). Reduction may be performed in *evaluation contexts*, defined in figure 2. Regarding these contexts, it is easy to see that the following holds.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

$$
\begin{array}{rcll}
M \to M' & \Rightarrow & [S \mid M] \to [S \mid M'] & \\
[S \mid \mathbf{E}[(\mathsf{ref}\, V)]] & \to & [u := V\,; S \mid \mathbf{E}[u]] & u \text{ fresh},\ \mathsf{fv}(V) \cap \mathsf{capt}(\mathbf{E}) = \emptyset \\
[S \mid \mathbf{E}[(!u)]] & \to & [S \mid \mathbf{E}[V]] & S(u) = V,\ \mathsf{fv}(V) \cap \mathsf{capt}(\mathbf{E}) = \emptyset \\
[S \mid \mathbf{E}[((\mathsf{set}\, u)V)]] & \to & [\{u := V\}S \mid \mathbf{E}[0]] & \mathsf{fv}(V) \cap \mathsf{capt}(\mathbf{E}) = \emptyset
\end{array}
$$

Fig. 4. Global reduction.

Remark
An evaluation context is either $[\,]$, or of the form $\mathbf{E}[\mathbf{F}]$ where $\mathbf{F}$ is a *frame*, given by the following grammar:

$$
\begin{array}{rcl}
\mathbf{F} & ::= & ([\,]N) \mid (V[\,]) \mid (\mathsf{let}\ x = [\,]\ \mathsf{in}\ M) \mid (\mathsf{let\ rec}\ x = [\,]\ \mathsf{in}\ M) \\
 & \mid & \langle [\,], \ell = N\rangle \mid \langle R, \ell = [\,]\rangle \mid ([\,].\ell) \mid ([\,]\backslash\ell)
\end{array}
$$

To describe the semantics of the imperative constructs, given by the rules for *global reduction* in figure 4, we enrich the language with a denumerable set $\mathcal{N}$ of *names*, or *locations* u, v, $w\ldots$, distinct from the variables and the labels. These names are also values. A *configuration* is a pair $[S \mid M]$ of an expression M and a *store* S, that is a mapping from locations to values. We use the following syntax for stores:

$$S ::= \varepsilon \mid u := V\,; S$$

The value $S(u)$ of a name in the store, and the partial operation $\{u := V\}S$ of updating the store, are defined in the obvious way. In the side conditions of the rules for global reduction, $\mathsf{capt}(\mathbf{E})$ is the set of variables that are captured by $\mathbf{E}$, that is, bound in $\mathbf{E}$ by a let rec binder introducing a sub-context. Let us see an example – which will be the standard object-oriented example of a "point". Assuming that some arithmetical operations are given, we define a "class" of unidimensional points as follows:

$$
\begin{array}{rcl}
\mathsf{let\ point} & = & \lambda x \lambda \mathsf{self} \langle \mathsf{pos} = \mathsf{ref}\, x, \\
 & & \qquad \mathsf{move} = \lambda y((\mathsf{set\ self.pos})(!\mathsf{self.pos} + y))\rangle\ \mathsf{in} \ldots
\end{array}
$$

Within the scope of this definition, we may define a point object, instance of that class, by intantiating the position parameter x to some initial value, and building a recursive record of methods. Let us define the fixpoint operator fix as follows:

$$\mathsf{fix} =_{\mathrm{def}} \lambda f(\mathsf{let\ rec}\ z = fz\ \mathsf{in}\ z)$$

Then, if we let $V = \lambda y((\mathsf{set}\ z.\mathsf{pos})(!z.\mathsf{pos} + y))$ and $R = \langle \mathsf{pos} = u, \mathsf{move} = V\rangle$, we have for instance:

$$
\begin{array}{rcl}
[\varepsilon \mid \mathsf{fix}(\mathsf{point}\,0)] & \stackrel{*}{\to} & [\varepsilon \mid (\mathsf{let\ rec}\ z = \langle \mathsf{pos} = \mathsf{ref}\,0, \mathsf{move} = V\rangle\ \mathsf{in}\ z)] \\
 & \to & [u := 0\,; \varepsilon \mid (\mathsf{let\ rec}\ z = R\ \mathsf{in}\ z)] \\
 & \stackrel{*}{\to} & [u := 0\,; \varepsilon \mid \langle \mathsf{pos} = u, \mathsf{move} = \{z \mapsto O\}V\rangle]
\end{array}
$$

where $O = (\mathsf{let\ rec}\ z = R\ \mathsf{in}\ R)$. One can see that there are two parts in this evaluated object: a state part, which records the (mutable) position of the object, and the (recursive, immutable) record of methods. Now imagine that we want to

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

enhance the point class with a clear method that resets the position to the origin. Then we introduce a new class inheriting from point:

$$\mathsf{let\ point}' = \lambda x \lambda \mathsf{self} \langle (\mathsf{point}\, x)\mathsf{self}\,,\ \mathsf{clear} = ((\mathsf{set\ self.pos})0) \rangle\ \mathsf{in} \ldots$$

This is a perfectly acceptable way of building a class inheriting from point, except that, due to the way the clear method is written, we cannot create an object instance of that class. More precisely, the type system will reject an expression like $\mathsf{fix}(\mathsf{point}'\,0)$, and rightly so. Indeed, if we try to compute this expression, we get stuck in $[u := 0\,; \varepsilon \mid (\mathsf{let\ rec}\ z = \mathbf{E}[z]\ \mathsf{in}\ z)]$ where $\mathbf{E} = \langle \mathsf{pos} = u, \mathsf{move} = V, \mathsf{clear} = ((\mathsf{set}\ [].\mathsf{pos})0) \rangle$. In the clear method, the self parameter ought to be protected from being evaluated, and a standard way to do this is to define this method as a "thunk", $\mathsf{clear} = \lambda y((\mathsf{set\ self.pos})0)$, which may be invoked on an object, $o = \mathsf{fix}(\mathsf{point}'\,42)$ for instance, as $o.\mathsf{clear}()$. This is the main technical point of the paper: to create objects instance of some class, we must be able to sometimes accept, sometimes reject expressions of the form $(\mathsf{let\ rec}\ z = (Gz)\ \mathsf{in}\ N)$, in particular when $G \xrightarrow{*} \lambda \mathsf{self} M$, depending on whether the function (with side effects) G is "protective" towards its argument or not. As we mentioned in the Introduction, unsafe recursion is not tied to record computations: it already shows up in the purely functional fragment of the language.

In the type system we will use a notion of a *pure* expression, which is an expression that can be evaluated without producing any side effect. In particular, evaluating a pure expression does not expand the store, and therefore such expressions have also been called non-expansive expressions. They are given by the following syntax:

$$\begin{array}{rcl} U & ::= & x \mid \lambda x M \mid (\mathsf{let}\ x = U\ \mathsf{in}\ U') \mid (\mathsf{let\ rec}\ x = U\ \mathsf{in}\ U') \\ & \mid & \langle U, \ell = U' \rangle \mid (U.\ell) \mid (U \backslash \ell) \end{array}$$

Lemma 2.2

(i) If U and U' are pure expressions, then $\{x \mapsto U\}U'$ is a pure expression.
(ii) If M is a pure expression and $M \to M'$ then M' is a pure expression.

Proof

(i) Immediate.
(ii) By induction on the proof of $M \to M'$, using the previous point. □

To establish a type safety result, we need to analyse the possible behaviour of expressions under evaluation: computing an expression may end on a value, or may go forever, but there are other possibilities – in particular an expression may "go wrong" (Milner, 1978) (or "be faulty", following the terminology of Wright & Felleisen (1994)). Our analysis is slightly non-standard here, since we have to deal with open terms. Besides the faulty expressions, we distinguish what we call "global redexes" and "head expressions". A global redex is an expression that has to be reduced, but whose reduction needs the participation of the store. A head expression is, by analogy with the "head normal forms" of the λ-calculus, a term M where a variable appears in the head position, that is $M = \mathbf{E}[x]$ for some evaluation context $\mathbf{E}$, and where something has to be done with the value of x.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

Definition 2.3
A term M is a *global redex* if M is $\mathbf{E}[(\mathsf{ref}\, V)]$, or $\mathbf{E}[(!u)]$, or else $\mathbf{E}[((\mathsf{set}\, u)V)]$ for some value V and location u, with $\mathsf{fv}(V) \cap \mathsf{capt}(\mathbf{E}) = \emptyset$.

Definition 2.4
A term M is a *head expression* if $M = \mathbf{H}[x]$ with $x \notin \mathsf{capt}(\mathbf{H})$, where the $\mathbf{H}$ contexts are given as follows:

$$\mathbf{H} ::= \mathbf{E}[([\,]\, V)] \mid \mathbf{E}[(!\,[\,])] \mid \mathbf{E}[(\mathsf{set}\,[\,])] \mid \mathbf{E}[([\,].\ell)] \mid \mathbf{E}[([\,]\backslash\ell)]$$

Definition 2.5
A term M is *faulty* if it contains a sub-expression of one of the following forms:

(i) (VN), where V is either a location, or $()$, or a record value;
(ii) $(\mathsf{let\ rec}\ x = \mathbf{H}[x]\ \mathsf{in}\ M)$ with $x \notin \mathsf{capt}(\mathbf{H})$
(iii) $(\mathsf{let\ rec}\ x = \mathbf{E}[N]\ \mathsf{in}\ M)$ where N is either $(\mathsf{ref}\, V)$ or $((\mathsf{set}\, u)V)$ with $x \in \mathsf{fv}(V)$;
(iv) $(!V)$ or $(\mathsf{set}\, V)$ where V is neither a variable nor a location;
(v) $\langle V, \ell = N\rangle$ where V is not a record value;
(vi) $(V.\ell)$ or $(V\backslash\ell)$, where V is neither a variable, nor a non-empty record-value.

Notice that an expression of the form $(\mathsf{set}\, V)$ where V is neither a variable nor a location is both a value and a faulty expression. This definition of faulty expressions extends the usual one (again, see Wright & Felleisen (1994)) with new cases regarding recursion: the evaluation of $(\mathsf{let\ rec}\ x = \mathbf{H}[x]\ \mathsf{in}\ M)$ is stuck if $x \notin \mathsf{capt}(\mathbf{H})$, since $\mathbf{H}[x]$ is not reducible, though not a value. The fact that $(\mathsf{let\ rec}\ x = \mathbf{E}[(\mathsf{ref}\, V)]\ \mathsf{in}\ M)$ and $(\mathsf{let\ rec}\ x = \mathbf{E}[((\mathsf{set}\, u)V)]\ \mathsf{in}\ M)$ have to be regarded as faulty when $x \in \mathsf{fv}(V)$ has to do with the way we have defined the operational semantics of recursive definitions, using substitution instead of environments: these expressions are global redexes, but they are not reducible since x is captured by the recursive definition. Then our first result is as follows.

Proposition 2.6
For any expression M, either M reduces, i.e. $M \to M'$ for some M', or M is a head expression, or a faulty expression, or a global redex, or a value.

Proof
By induction on the structure of M. The case where M is a value is trivial.

1. In the case of an application (MN), we use the induction hypothesis for M:

1.1. if M reduces into M' then (MN) reduces into $(M'N)$.
1.2. If M is a head expression, then (MN) is also a head expression.
1.3. If M is faulty, then (MN) is faulty too.
1.4. If M is a global redex, then the same holds for (MN).
1.5. If M is a value V, we examine the possible cases for V. If V is neither a variable, nor a functional value (that is, V is either a location, or $()$, or a record value), then (VN) is a faulty expression. If V is a variable or a functional value, we use the induction hypothesis for N. If N reduces, or is a head expression, or is faulty, or else is a global redex, then the same holds for (VN). If N is a value W, then we examine the possible cases for V: if V is a variable, then (VW) is a head

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

expression. If $V = \mathsf{ref}$ then (VW) is a global redex. If $V = !$ then either W is not a variable, nor a location, in which case (VW) is faulty, or (VW) is a global redex (if W is a location) or a head expression (if W is a variable). If $V = \mathsf{set}$ then (VW) is a value (which may be faulty if W is neither a variable nor a location, and is also a head expression if W is a variable). If $V = (\mathsf{set}\, V')$ then (VW) is a head expression if V' is a variable, or a global redex is V' is a location, and a faulty expression otherwise. Finally, if V is an abstraction $\lambda x M'$, then (VW) reduces to $\{x \mapsto W\}M'$.

2. In the case of a let expression $(\mathsf{let}\ x = N\ \mathsf{in}\ M)$, we use the induction hypothesis for N:
2.1–4. if N reduces, or is a head expression, or is faulty, or else is a global redex, then the same holds for $(\mathsf{let}\ x = N\ \mathsf{in}\ M)$.
2.5. If N is a value V, then $(\mathsf{let}\ x = N\ \mathsf{in}\ M)$ reduces to $\{x \mapsto V\}M$.

3. In the case of a let expression $(\mathsf{let\ rec}\ x = N\ \mathsf{in}\ M)$, we use the induction hypothesis for N:
3.1. if N reduces, then $(\mathsf{let\ rec}\ x = N\ \mathsf{in}\ M)$ reduces too.
3.2. If N is a head expression $\mathbf{H}[z]$, there are two cases: if $z = x$ then $(\mathsf{let\ rec}\ x = N\ \mathsf{in}\ M)$ is faulty, and a head expression otherwise.
3.3. If N is faulty, then $(\mathsf{let\ rec}\ x = N\ \mathsf{in}\ M)$ is faulty.
3.4. If N is a global redex $\mathbf{E}[N']$, then either there exist V and $x \in \mathsf{fv}(V)$ such that $N' = (\mathsf{ref}\, V)$ or $N' = ((\mathsf{set}\, u)V)$, in which cases $(\mathsf{let\ rec}\ x = N\ \mathsf{in}\ M)$ is faulty, or $(\mathsf{let\ rec}\ x = N\ \mathsf{in}\ M)$ is a global redex.
3.5. If N is a value V, then $(\mathsf{let\ rec}\ x = N\ \mathsf{in}\ M)$ reduces to $\{x \mapsto (\mathsf{let\ rec}\ x = V\ \mathsf{in}\ V)\}M$.

All the other cases, that is $\langle M, \ell = N\rangle$, $(M.\ell)$, $(M \backslash \ell)$, are similar to the one of the application (MN) (and in fact simpler for the latter two). □

Notice that in the last case of proof (3.5), we may have $N = x$, or $N = \langle x, \ell = W\rangle$, or else $N = \langle R, \ell = x\rangle$. Although the expression $(\mathsf{let\ rec}\ x = N\ \mathsf{in}\ M)$ is not considered faulty in these cases – its evaluation may diverge, depending on how x is used in M –, we will see that it is rejected by the type system (see Corollary 3.3).

Let us say that a configuration $[S \mid M]$ is *closed* if M is a closed expression, that is $\mathsf{fv}(M) = \emptyset$, $S = u := V\,; S'$ implies that V is closed, and the locations occurring in M are in $\mathsf{dom}(S)$. It is easy to see that if $[S \mid M] \to [S' \mid N]$ and $[S \mid M]$ is closed, then $[S' \mid N]$ is closed too.

Corollary 2.7
For any closed configuration $[S \mid M]$, if its evaluation terminates on $[S' \mid N]$, that is $[S \mid M] \overset{*}{\to} [S' \mid N]$, and $[S' \mid N]$ is irreducible, then N is either faulty or a value.

3 The type system

The aim in using a type system is to prevent run-time errors – and also to provide some interesting information about expressions. Then we have to design such a system in a way that rules out faulty expressions. Apart from the case of let rec

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

expressions, that is (ii) and (iii) of Definition 2.5, we already know how to do that: use functional types $\theta \to \tau$ for expressions that have to be applied, reference types $\tau\,\mathsf{ref}$ for expressions that have to be de-referenced or assigned to, and record types, with row variables (Rémy, 1994b; Wand, 1987), $\langle \rho, \ell_1 : \tau_1, \ldots, \ell_k : \tau_k \rangle$ for expressions that are subject to selection, or extension, or restriction. Since we want to be able to define recursive expressions of any type, there is no specific type construct associated with recursion. However, to exclude unsafe recursion, we will use "decorated types", where the decorations are boolean values 0 or 1 (with $0 \leqslant 1$), also called safeness degrees, to which we must add, in order to obtain principal types, degree variables $p, q \ldots$ We denote by $a, b, c \ldots \in \mathscr{D}$ these *degrees*, either constant or variable.

Following Milner (1978), we use a polymorphic let construct. This is crucial for defining classes that may be inherited in various ways, and instantiated into objects, and to type generic functions like $\lambda x(x.\ell)$ or $\lambda x \lambda y \langle x, \ell \leftarrow y \rangle$. Then we will use *type schemes*. As in Jategaonkar & Mitchell (1993), we do not allow the same label to occur several times in a given record type – but our treatment of row variables is quite different from the one of Jategaonkar & Mitchell (1993). Therefore, in quantifying on a row variable, we must take into account the context in which it occurs, by means of the set L of labels that it must not contain. More precisely, we use (series of) quantifications of the form $(\forall t :: L.\sigma)$, meaning that the type variable t ranges over types which are not record types containing a label mentioned in L. This is similar to the bounded quantification $(\forall t <: \Diamond \backslash L.\sigma)$ of Cardelli & Mitchell (1994), but notice that, apart from "generic instantiation" (see below), we do not use any form of subtyping here. In Jategaonkar & Mitchell (1993), annotated type variables, that is t_L, are used for the same pupose. Given a set $\mathscr{T}y\mathscr{V}ar$ of type variables, the syntax of types and type schemes is:

$$\begin{array}{rcl} \tau, \theta \ldots & ::= & \mathsf{unit} \mid t \mid (\theta^a \to \tau) \mid \tau\,\mathsf{ref} \mid \rho \\ \rho & ::= & t \mid \Diamond \mid \langle \rho, \ell : \tau \rangle \\ \sigma, \zeta \ldots & ::= & \tau \mid (\forall Q.\sigma) \\ Q & ::= & t :: L \mid t :: L, Q \end{array}$$

where t is any type variable, a is any degree, and L is any finite set of labels. As for record expressions, we shall denote a record type $\langle \cdots \langle \Diamond, \ell_1 : \tau_1 \rangle \ldots, \ell_k : \tau_k \rangle$ by $\langle \ell_1 : \tau_1 \ldots, \ell_k : \tau_k \rangle$, and similarly $\langle t, \ell_1 : \tau_1 \ldots, \ell_k : \tau_k \rangle$ stands for the "open" record type $\langle \cdots \langle t, \ell_1 : \tau_1 \rangle \ldots, \ell_k : \tau_k \rangle$. We shall denote by $=_{\mathscr{T}}$ the least congruence on type schemes containing α-conversion of type variables bound by quantification, and satisfying the following law:

$$\langle \langle \rho, \ell : \tau \rangle, \ell' : \tau' \rangle = \langle \langle \rho, \ell' : \tau' \rangle, \ell : \tau \rangle$$

As we said, not all record types are legal types: for instance $\langle \langle \rho, \ell : \tau \rangle, \ell' : \tau' \rangle$ is not regarded as a well-formed type if $\ell' = \ell$. Then we have a simple inference system allowing us to infer judgements of the form $Q \vdash \sigma :: L$, which can be read "σ is well-formed under the assumption Q, and is not a record type containing one of the labels in L".

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

$$\frac{}{t :: L, C \vdash t :: L'}\ L' \subseteq L \qquad \frac{}{C \vdash \mathsf{unit} :: \emptyset} \qquad \frac{C \vdash \theta :: \emptyset \quad C \vdash \tau :: \emptyset}{C \vdash (\theta^a \to \tau) :: \emptyset}$$

$$\frac{C \vdash \tau :: \emptyset}{C \vdash \tau\ \mathsf{ref} :: \emptyset} \qquad \frac{}{C \vdash \langle\rangle :: L} \qquad \frac{C \vdash \rho :: L \cup \{\ell\} \quad C \vdash \tau :: \emptyset}{C \vdash \langle \rho, \ell : \tau \rangle :: L}\ \ell \notin L$$

$$\frac{Q, C \vdash \sigma :: \emptyset}{C \vdash (\forall Q.\sigma) :: \emptyset}\ \mathsf{dom}(Q) \cap \mathsf{dom}(C) = \emptyset$$

$$\frac{}{C \vdash 0 \leqslant \alpha} \qquad \frac{}{C \vdash \alpha \leqslant 1} \qquad \frac{}{C \vdash \alpha \leqslant \alpha} \qquad \frac{C \vdash \alpha \leqslant \kappa \quad C \vdash \kappa \leqslant \beta}{C \vdash \alpha \leqslant \beta}$$

$$\frac{}{p \leqslant \alpha, C \vdash p \leqslant \alpha} \qquad \frac{}{C \vdash (\alpha \wedge \beta) \leqslant \alpha} \qquad \frac{}{C \vdash (\alpha \wedge \beta) \leqslant \beta} \qquad \frac{C \vdash \kappa \leqslant \alpha \quad C \vdash \kappa \leqslant \beta}{C \vdash \kappa \leqslant (\alpha \wedge \beta)}$$

Fig. 5. Constraints: annotations and inequalities.

The annotation of type variables with finite sets of labels is not the only constraint we have to take into account in the type system: we also have to deal with constraints on degrees, that take the form of a set of inequalities $p \leqslant \alpha$ where p is a degree variable and α is a *degree expression*, built from degrees by using the conjunction (that is, the meet) operation $\wedge$. We denote by $\alpha, \beta, \kappa \ldots \in \mathscr{D}\mathscr{E}xp$ these expressions. The meet operation is used to represent conjunction of constraints; namely, $p \leqslant \alpha \wedge \beta$ is equivalent to $p \leqslant \alpha$ & $p \leqslant \beta$. Notice that constraints on degrees of this kind are obviously satisfiable, e.g. assigning uniformly 0 to the degree variables. In order to give a simple form to typing rules, we group the two kinds of constraints – type variables annotations $t :: L$ and inequalities $p \leqslant \alpha$ – into a single component, called a *constraint*, denoted by C. This is a pair of a mapping C_{typ} from a finite set $\mathsf{dom}(C_{\mathsf{typ}})$ of type variables into annotations (finite subsets of $\mathscr{L}$), and of a mapping C_{deg} from a finite set $\mathsf{dom}(C_{\mathsf{deg}})$ of degree variables into degree expressions. As usual, we write $t :: L, C$ for the constraint C updated by the assignment of annotation L to t (so that a series of annoted type variables, $Q = t_1 :: L_1, \ldots, t_n :: L_n$ is regarded as a constraint), and similarly for $p \leqslant \alpha, C$. This notation is extended to C, C' in the obvious way. The constraint system, given in figure 5, allows us to infer judgements of the form $C \vdash \sigma :: L$, as well as $C \vdash \alpha \leqslant \beta$, meaning that this inequality is a consequence of C. As one can see, the only constraints on types are that in $\langle \rho, \ell : \tau \rangle$, the type ρ must not contain the label ℓ, and that one may only quantify over type variables which do not still appear in the context. Notice that if $C \vdash \sigma :: L$ and σ is not a record type, then $L = \emptyset$. Given a set A of assertions of the form $\sigma :: L$ or $\alpha \leqslant \beta$, we denote by $C \vdash A$ the fact that all the assertions of A are provable from the constraint C.

As usual, we need the notion of an instance of a type scheme, obtained by substituting not only types for type variables, but also degrees for degree variables. Then a type and degree substitution S is a mapping from type variables to types, and from degree variables to degrees (not degree expressions), which is the identity, except

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

for a finite set $\mathsf{dom}(\mathsf{S})$ of variables. We write $\mathsf{S} = \{ t_i \mapsto \tau_i \mid i \in I \} \cup \{ p_j \mapsto a_j \mid j \in J \}$ if $\mathsf{S}(t_i) = \tau_i$, $\mathsf{S}(p_j) = a_j$, and S is the identity otherwise. If X is a set of (type and degree) variables, and S is a substitution, then $\mathsf{S} \restriction X$ is the substitution that coincides with S on X, and is the identity otherwise. We denote by $\mathsf{S}(\sigma)$ the result of applying the (capture-free) substitution S to the type scheme σ, and similarly for $\mathsf{S}(\alpha)$. The composition of substitutions is denoted $\mathsf{S}'\mathsf{S}$, with $\mathsf{S}'\mathsf{S}(t) = \mathsf{S}'(\mathsf{S}(t))$. In most cases, we need to ensure that applying a substitution to a type scheme results in a well-formed type. For instance, applying $\{t \mapsto \theta\}$ to $\langle t, \ell : \tau \rangle$ yields a syntax error if θ is not a record type, and a non well-formed type if θ is a record type that contains the label ℓ. Given two constraints C_0 and C_1, we then define $\mathscr{S}ub(C_0, C_1)$ as follows:

$$\mathsf{S} \in \mathscr{S}ub(C_0, C_1) \Leftrightarrow_{\text{def}} \mathsf{dom}(\mathsf{S}) \subseteq \mathsf{dom}(C_0) \ \&\ C_1 \vdash \mathsf{S}(C_0)$$

where $\mathsf{S}(C) = \{ \mathsf{S}(t) :: L \mid t :: L \in C \} \cup \{ \mathsf{S}(p) \leqslant \mathsf{S}(\alpha) \mid p \leqslant \alpha \in C \}$. Then it is easy to see, by induction on σ, that the following holds.

Lemma 3.1
If $C_0 \vdash \sigma :: L$ and $\mathsf{S} \in \mathscr{S}ub(C_0, C_1)$ then $C_1 \vdash \mathsf{S}(\sigma) :: L$.

Then, for instance, the standard relation of being a *generic instance* (see Damas & Milner (1982)) is relative to some constraint: we write $C \vdash \sigma \geq \sigma'$ if $\sigma = (\forall Q.\tau)$ and $\sigma' = (\forall Q'.\mathsf{S}(\tau))$ for some $\mathsf{S} \in \mathscr{S}ub(Q, C \cup Q')$.

The typing judgements have the form $C \,;\, \Gamma^\gamma \vdash M : \tau$, where C is a constraint, τ is a type([1]), and Γ^γ is a *typing context*. This is a pair of two maps Γ and γ, respectively from a finite set of variables and locations to type schemes (for variables) and types (for locations), and from a finite set of variables to degree expressions. The idea is that with a variable x we associate an assumption about the fact that it will or will not occur in a dangerous – w.r.t. recursion – position. This assumption is the *safeness degree* of the variable in the context – 0 standing for "dangerous", i.e. potentially unsafe. We call Γ a *type assumption* and γ a *degree assumption* (or sometimes a degree assignment). We assume that Γ and γ assign type schemes and degree expressions to the same variables, that is $\mathsf{dom}(\gamma) = \mathsf{dom}(\Gamma) \cap \mathscr{X}$. Denoting a pair (σ, α) by σ^α, we will write the typing context Γ^γ as

$$u_1 : \tau_1, \ldots, u_k : \tau_k, x_1 : \sigma_1^{\alpha_1}, \ldots, x_n : \sigma_n^{\alpha_n}$$

if $\Gamma = \{u_1 \mapsto \tau_1, \ldots, u_k \mapsto \tau_k, x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\}$ and $\gamma = \{x_1 \mapsto \alpha_1, \ldots, x_n \mapsto \alpha_n\}$. We use the following predicate on degree assumptions:

$$\begin{array}{c} \text{let } X \text{ be a set of variables. Then } C \vdash \delta \leqslant \gamma \text{ on } X \text{ if and only if} \\ C \vdash \delta(x) \leqslant \gamma(x) \text{ for all } x \in X. \end{array}$$

In the type system, we abbreviate $\lambda x(\mathsf{if}\ x \in \mathsf{fv}(M)\ \mathsf{then}\ \alpha\ \mathsf{else}\ 1)$ into α_M. As usual, we let $\delta \wedge \gamma$ denote the function defined pointwise, by $(\delta \wedge \gamma)(x) = \delta(x) \wedge \gamma(x)$. We also abusively write 1 for $\lambda x.1$, and similarly for 0. In the typing axioms, we have

[1] To simplify the presentation we do not include the usual rules of instantiation and generalization (see Damas & Milner (1982)), but they would easily be shown to be admissible if judgements $C \,;\, \Gamma^\gamma \vdash M : \sigma$ were allowed, and therefore we will use them in the examples.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

$$\frac{C\,;\Gamma^{\gamma}\vdash M:\tau \qquad C\vdash \delta\leqslant\gamma \text{ on } \mathsf{fv}(M)}{C\,;\Gamma^{\delta}\vdash M:\tau} \qquad \frac{C\vdash x:\sigma,\Gamma \qquad C\vdash \sigma\geq\tau}{C\,;x:\sigma^{0},\Gamma^{1}\vdash x:\tau}$$

$$\frac{C\,;x:\theta^{a},\Gamma^{\gamma}\vdash M:\tau}{C\,;\Gamma^{1}\vdash \lambda xM:(\theta^{a}\to\tau)} \qquad \frac{C\,;\Gamma^{\gamma}\vdash M:\theta^{a}\to\tau \qquad C\,;\Gamma^{\gamma}\vdash N:\theta}{C\,;\Gamma^{0_M\wedge\delta}\vdash (MN):\tau} \quad (1)$$

$$\frac{Q,C\,;\Gamma^{\gamma}\vdash N:\theta \qquad C\,;x:(\forall Q.\theta)^{\alpha},\Gamma^{\gamma}\vdash M:\tau}{C\,;\Gamma^{\delta}\vdash (\mathsf{let}\ x=N\ \mathsf{in}\ M):\tau} \quad (2)$$

$$\frac{Q,C\,;x:\theta^{1},\Gamma^{\gamma}\vdash N:\theta \qquad C\,;x:(\forall Q.\theta)^{\alpha},\Gamma^{\gamma}\vdash M:\tau}{C\,;\Gamma^{\delta}\vdash (\mathsf{let\ rec}\ x=N\ \mathsf{in}\ M):\tau} \quad (2)$$

(1) where

$$\delta(x)=\begin{cases} a & \text{if } N=x\\ (a_N\wedge\gamma)(x) & \text{otherwise}\end{cases}$$

(2) where $t\in\mathsf{dom}(Q)\ \Rightarrow\ t\notin\mathsf{dom}(C)$ and Q is empty if N is not pure, and

$$\delta\ =\begin{cases}\alpha_N\wedge\gamma & \text{if } M \text{ is not pure}\\ \gamma & \text{otherwise}\end{cases}$$

Fig. 6. The type system (functional fragment).

to check that the types involved in the judgement are acceptable. This is done by means of a proof system for judgements of the form $C\vdash\Gamma$, which is actually trivial:

$$\frac{}{C\vdash\emptyset} \qquad \frac{C\vdash\sigma::\emptyset \qquad C\vdash\Gamma}{C\vdash x:\sigma,\Gamma}$$

Now let us comment on some of the rules that are presented in figures 6 and 7. The first one is a "degree weakening" rule, stating that "optimistic" assumptions, assigning for instance degree 1 to some variables, can always be safely downgraded. The intuition about degrees is that a variable is safe, and therefore may be assigned degree 1, basically when it occurs within a value, that is guarded by a λ-abstraction, and more generally when its specific value is not needed during the computation (see Corollary 4.10). This explains the typing axiom $C\,;x:\sigma^{0},\Gamma^{1}\vdash x:\tau$: a value for x must be fetched, say, from the environment, to evaluate the expression x, and therefore x has degree 0. On the other hand, any other variable is not concerned with the evaluation of x, hence may be assigned degree 1. More generally, in all typing axioms we make such "optimistic" degree assumptions, wich can always be weakened, regarding the variables that do not occur in the typed expression.

In the rule for abstraction of x, we assume that the degree of x does not contain the $\wedge$ operation, but this is not a restriction, since we may always add a fresh constraint $p\leqslant\alpha$ and use the weakening rule. The rule for abstraction promotes the typing context to a definitely safe one (Γ^{1}), since all the variables occurring in the abstraction value are protected from being evaluated by the λ. Conversely, the variables occurring in the function part of an application are potentially dangerous,

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

like for instance x in $(\lambda y.xy)V$. Then they are all downgraded to having the degree 0. Regarding the argument, we must be more careful: applying a function of type $\theta^1 \to \tau$ that does not put its argument in danger, like $\lambda f \lambda x(fx)$ for instance (see below for the typing of this expression), we may decide that its free variables are protected. However, this is only true if they are protected in the argument itself. Then applying a function of type $\theta^a \to \tau$ places the argument in a position where the variables have a degree which is, at best, a or the degree they have in the argument. This is where we use the $\wedge$ operation([2]). Nevertheless, we are able to make a special case when the argument is a variable x (whose degree is 0): anticipating that the function M is an abstraction $\lambda y M'$, thus typed using an assumption about y, we may consider that the degree of x in (Mx) is the one of y in M', that is a, instead of $a \wedge 0 = 0$ (see Lemma 4.5). This is important for our purpose, where we should not always reject (let rec $x = Gx$ in M), see Section 6. One could probably extend this specific treatment of the argument to the case where it is a value, but this does not seem to be a good idea, because then we could not predict how the recursive variables are used (see Corollary 4.10). Let us see an example. Since

$$\dfrac{\dfrac{\vdots}{C \vdash \alpha \leqslant \beta} \qquad \dfrac{}{C \vdash \alpha \leqslant 1}}{C \vdash \alpha \leqslant \beta \wedge 1}$$

the term $\lambda f \lambda x(fx)$ has the following typing, where $C = t :: \emptyset, t' :: \emptyset, q \leqslant p$ (omitting the proof of $C \vdash f : (t^p \to t'), x : t$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{C\,;f:(t^p \to t')^0, x:t^1 \vdash f:(t^p \to t')} \qquad \dfrac{\vdots}{C\,;f:(t^p \to t')^1, x:t^0 \vdash x:t}}{C\,;f:(t^p \to t')^0, x:t^0 \vdash f:(t^p \to t')} \qquad \dfrac{\vdots}{C\,;f:(t^p \to t')^0, x:t^0 \vdash x:t}}{C\,;f:(t^p \to t')^{0\wedge 1}, x:t^{1\wedge p} \vdash (fx):t'}}{C\,;f:(t^p \to t')^0, x:t^q \vdash (fx):t'}}{C\,;f:(t^p \to t')^1 \vdash \lambda x(fx):t^q \to t'}}{C;\vdash \lambda f \lambda x(fx):(t^p \to t')^1 \to t^q \to t'}}{q \leqslant p;\vdash \lambda f \lambda x(fx):(\forall t :: \emptyset, t' :: \emptyset.(t^p \to t')^1 \to t^q \to t')}$$

In the examples that follow, we shall often omit the use of the degree weakening rule, when it amounts to use obvious inequalities like $\alpha \leqslant \alpha \wedge 1$. To see why we need the meet operation, the reader may try to type $f(gx)$, where the degree of x depends on the nature of both f and g. As we said in the introduction, we call *protective*

[2] For simplicity, our presentation of the type system is "additive", in the sense that the components of a binary construct share the same typing context. However, we may have to use the degree weakening rule, hence the meet, to keep to this pattern.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

a function which has a type of the form $\theta^1 \to \tau$. Notice however that this is not necessarily the most general type of the function. For instance, the term $\mathsf{K} = \lambda x \lambda y\, x$ has, as we shall see, most general type $t_0^p \to t_1^q \to t_0$ (without any constraint), which can be instantiated into $\tau^1 \to \theta^a \to \tau$. One may also observe that η-expansion is of no help in building protective functions: indeed, it is easy to see that if $\lambda x(Mx)$ has type $\theta^a \to \tau$, then M has the same type. Moreover, one cannot generally η-expand recursive variables, because they may not have a functional type.

One may notice that in the rule for the let construct, we use the "value polymorphism" approach of SML (Milner *et al.*, 1997) (also proposed by Wright (1995)), to solve the difficulties in polymorphic typing due to imperative features. The reason why we make a special case when M is pure will be explained in Section 6. The rule for the $\mathsf{let\,rec}$ construct is the only one involving a real (i.e. possibly unsatisfiable) constraint on degrees, namely $1 \leqslant \alpha$. It is exemplified by the following typing of the fixpoint combinator, where $\Gamma = \{f \mapsto (t^1 \to t), x \mapsto t\}$, $\gamma = \{f \mapsto 0, x \mapsto 0\}$ and $\delta = \{f \mapsto 0, x \mapsto 1\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{t :: \emptyset\,;\, \Gamma^\gamma \vdash f : (t^1 \to t)} \quad \dfrac{\vdots}{t :: \emptyset\,;\, \Gamma^\gamma \vdash x : t}}{t :: \emptyset\,;\, \Gamma^\delta \vdash fx : t} \quad \dfrac{\vdots}{t :: \emptyset\,;\, \Gamma^\gamma \vdash x : t}}{t :: \emptyset\,;\, f : (t^1 \to t)^0 \vdash (\mathsf{let\ rec}\ x = fx\ \mathsf{in}\ x) : t}}{t :: \emptyset;\, \vdash \mathsf{fix} : (t^1 \to t)^0 \to t}$$

Notice that, as in ML, $(\mathsf{let\ rec}\ f = \lambda x N\ \mathsf{in}\ M)$ is always allowed, provided that M and N have appropriate typings. For instance, the call-by-value fixpoint combinator

$$\mathsf{Y} = (\mathsf{let\ rec}\ y = \lambda f.f(\lambda x.yfx)\ \mathsf{in}\ y)$$

has the following typing:

$$t :: \emptyset, t' :: \emptyset, p \leqslant q;\, \vdash \mathsf{Y} : ((t^p \to t')^r \to t^q \to t')^0 \to t^q \to t'$$

An example of a program from the "functional ML" fragment of the language that is rejected by our type system – as any other faulty expression, as we shall see –, is $(\mathsf{let\ rec}\ x = \mathsf{F}(xV)\ \mathsf{in}\ x)$ where F is the combinator $\lambda z \lambda y\, y$. Observe that this expression would be typable – provided that V has some type – with a "standard" typing rule for $\mathsf{let\,rec}$ expressions, that is omitting the constraint on the degree of the recursive variable, but that it is an irreducible expression which is not a value. Conversely, types are, as usual, only approximations, and some expressions are rejected that actually do not cause any trouble. For instance, assigning degree 0 to variables that appear in a function position, that is at the left of an application, is sometimes overly pessimistic, as in $(\mathsf{let\ rec}\ x = (\mathsf{F}x)V\ \mathsf{in}\ \cdots)$ where $x \notin \mathsf{fv}(V)$ for instance. Some other examples are given below (see the comments on the proof of Proposition 4.6).

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

$$\frac{C \vdash u : \tau, \Gamma}{C\,;\, u : \tau, \Gamma^1 \vdash u : \tau\,\mathsf{ref}} \qquad \frac{C \vdash \Gamma \qquad C \vdash \tau :: \emptyset}{C\,;\, \Gamma^1 \vdash \mathsf{ref} : \tau^0 \to \tau\,\mathsf{ref}} \qquad \frac{C \vdash \Gamma \qquad C \vdash \tau :: \emptyset}{C\,;\, \Gamma^1 \vdash \mathbf{!} : (\tau\,\mathsf{ref})^0 \to \tau}$$

$$\frac{C \vdash \Gamma \qquad C \vdash \tau :: \emptyset}{C\,;\, \Gamma^1 \vdash \mathsf{set} : (\tau\,\mathsf{ref})^0 \to \tau^0 \to \mathsf{unit}} \qquad \frac{C \vdash \Gamma}{C\,;\, \Gamma^1 \vdash () : \mathsf{unit}}$$

$$\frac{C \vdash \Gamma}{C\,;\, \Gamma^1 \vdash \langle\rangle : \langle\rangle} \qquad \frac{C\,;\, \Gamma^\gamma \vdash M : \rho \qquad C\,;\, \Gamma^\gamma \vdash N : \tau \qquad C \vdash \rho :: \{\ell\}}{C\,;\, \Gamma^\gamma \vdash \langle M, \ell = N\rangle : \langle \rho, \ell : \tau\rangle}$$

$$\frac{C\,;\, \Gamma^\gamma \vdash M : \langle \rho, \ell : \tau\rangle}{C\,;\, \Gamma^\gamma \vdash (M.\ell) : \tau} \qquad \frac{C\,;\, \Gamma^\gamma \vdash M : \langle \rho, \ell : \tau\rangle}{C\,;\, \Gamma^\gamma \vdash (M \backslash \ell) : \rho}$$

$$\frac{C \vdash \Gamma}{C\,;\, \Gamma \vdash \varepsilon} \qquad \frac{C\,;\, u : \tau, \Gamma^\gamma \vdash V : \tau \qquad C\,;\, u : \tau, \Gamma \vdash S}{C\,;\, u : \tau, \Gamma \vdash u := V\,;\, S} \qquad \frac{C\,;\, \Gamma \vdash S \qquad C\,;\, \Gamma^\gamma \vdash M : \tau}{C\,;\, \Gamma \vdash [S \mid M] : \tau}$$

Fig. 7. The type system (continued).

The functional core of the language concentrates all the subtleties of the use of degrees – the rest of the type system is quite trivial, and in particular there is not much choice in the typing of the record constructs: as we said, the extension operation $\langle M, \ell = N\rangle$ is strict, and thus requires that M is a record not containing the label ℓ, and similarly the strict restriction operation $(M \backslash \ell)$ requires ℓ to be present in M. Then there is no ambiguity in typing these operations. We must point out that, in the typing rules for records, we implicitly use the type equality relation $=_{\mathscr{T}}$. Admittedly, having to check the well-formedness of record types complicates the system, in a way that has nothing to do with the problem of typing safe recursion, but we think that solving this problem is only worth if this has some application, and that an interesting one is in modelling object-oriented constructions.

To conclude this section, we show that in a typable expression, recursive variables can only be "passed around", as arguments of protective functions – unless they are embedded within abstractions. To this end, we first need a technical result, showing that evaluation contexts are generally not "protective".

Lemma 3.2

Let N be an expression which is not a variable. If $C\,;\, \Gamma^\gamma \vdash N : \tau$ implies $C \vdash \gamma(x) \leqslant 0$, and if $C'\,;\, \Delta^\delta \vdash \mathbf{E}[N] : \theta$ with $x \notin \mathsf{capt}(\mathbf{E})$, then $C' \vdash \delta(x) \leqslant 0$.

Proof

By induction on $\mathbf{E}$. This is trivial for $\mathbf{E} = []$. Otherwise, we have $\mathbf{E} = \mathbf{E}'[\mathbf{F}]$ where $\mathbf{F}$ is a frame (see Remark 2.1). Then one can check, by cases on $\mathbf{F}$, that $C''\,;\, \Sigma^\xi \vdash \mathbf{F}[N] : \tau'$ implies $C'' \vdash \xi(x) \leqslant 0$. The hypothesis that N is not a variable is used in the case where $\mathbf{F} = (V\,[])$. □

Corollary 3.3

If $(\mathsf{let\ rec}\ x = \mathbf{E}[x]\ \mathsf{in}\ M)$ is typable, then $\mathbf{E} = \mathbf{E}'[(V\,[])]$ where V is a protective function.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

Proof
If (let rec $x = \mathbf{E}[x]$ in M) is typable, then $\mathbf{E}[x]$ is typable in a typing context where the degree of x is 1. This implies $\mathbf{E} \neq []$. If $\mathbf{E} = \mathbf{E}'[\mathbf{F}]$, then the previous lemma, where we let $N = \mathbf{F}[x]$, shows that we can only have $\mathbf{F} = (V\,[])$. Moreover, the degree of x in typing (Vx) must be 1, and therefore V is protective. □

4 Type safety

In this section we prove our first technical result, asserting that typable programs cannot "go wrong", that is they do not entail any run-time error. A first step towards this property is the following.

Lemma 4.1
The faulty expressions are not typable.

Proof
The type system is compositional, and therefore a term is typable only if all its subterms are typable. Then it is enough to check that the "basic" faulty expressions, as defined by the clauses (i)–(vi) of Definition 2.5 are not typable. The cases of (i) and (iv)–(vi) are immediate. Regarding (ii), one first notices that if $C\,;\Gamma^\gamma \vdash N : \tau$ where N is either (xV), or $(!\,x)$, or $(\mathsf{set}\,x)$, or else $(x.\ell)$ or $(x\backslash\ell)$, then $C \vdash \gamma(x) \leqslant 0$. Then, by Lemma 3.2, $C\,;\Gamma^\gamma \vdash \mathbf{H}[x] : \tau$ implies $C \vdash \gamma(x) \leqslant 0$ if $x \notin \mathsf{capt}(\mathbf{H})$. Then (let rec $x = \mathbf{H}[x]$ in M) is not typable, since this would imply $C \vdash 1 \leqslant 0$, which is impossible, for C is consistent. The proof is similar in the case (iii). □

Then we have the standard "type preservation" – or "subject reduction" – property. To establish this property, we need some preliminary results. First we observe that if $C\,;\Gamma^\gamma \vdash M : \tau$ is provable, then $C \vdash \Gamma$ and $C \vdash \tau :: \emptyset$. The following weakening property is standard.

Lemma 4.2 (Weakening)

(i) If $C\,;\Gamma^\gamma \vdash M : \tau$ then for any x and σ such that $C \vdash \sigma :: \emptyset$ the judgement $C\,;\Gamma^\gamma, x : \sigma^\alpha \vdash M : \tau$ is provable, with a proof having the same structure as the one of $C\,;\Gamma^\gamma \vdash M : \tau$.
(ii) If $C\,;\Gamma^\gamma \vdash M : \tau$ then for any u and θ such that $C \vdash \theta :: \emptyset$ the judgement $C\,;\Gamma^\gamma, u : \theta \vdash M : \tau$ is provable, with a proof having the same structure as the one of $C\,;\Gamma^\gamma \vdash M : \tau$.
(iii) If $C\,;\Gamma^\gamma \vdash M : \tau$ and C' is a constraint such that $\mathsf{dom}(C') \cap \mathsf{dom}(C) = \emptyset$, then $C', C\,;\Gamma^\gamma \vdash M : \tau$ is provable, with a proof having the same structure as the one of $C\,;\Gamma^\gamma \vdash M : \tau$.

We omit the similar statements regarding the judgements $C\,;\Gamma \vdash S$ and $C\,;\Gamma \vdash [S \mid M] : \tau$. Then, given a substitution $\mathsf{S} \in \mathscr{S}ub(C_0, C_1)$, a typing context Γ^γ and a type τ such that $C_0 \vdash \Gamma$ and $C_0 \vdash \tau :: \emptyset$, we denote by $\mathsf{S}(\Gamma^\gamma \vdash M : \tau)$ the statement $\Delta^\delta \vdash M : \mathsf{S}(\tau)$ where $\Delta = \mathsf{S} \circ \Gamma$ and $\delta = \mathsf{S} \circ \gamma$. We have the standard result that typing is compatible with type (and degree) substitution (see Damas & Milner (1982)).

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

Lemma 4.3
If $C\,;\Gamma^\gamma \vdash M:\tau$ and $\mathsf{S}\in \mathscr{S}ub(C,C')$ then $C'\,;\mathsf{S}(\Gamma^\gamma \vdash M:\tau)$ is provable, with a proof having the same structure as the one of $C\,;\Gamma^\gamma \vdash M:\tau$.

Proof
By induction on the inference of $C\,;\Gamma^\gamma \vdash M:\tau$, straightforward. □

The next properties are kind of converse of the weakening ones.

Lemma 4.4
(i) If $C\,;x:\sigma^\alpha,\Gamma^\gamma \vdash M:\tau$ and $x\notin \mathsf{fv}(M)$ then $C\,;\Gamma^\gamma \vdash M:\tau$ is provable, with a proof having the same structure as the one of $C\,;x:\sigma^\alpha,\Gamma^\gamma \vdash M:\tau$.
(ii) If $C\,;u:\theta,\Gamma^\gamma \vdash M:\tau$ and u does not occur in M then $C\,;\Gamma^\gamma \vdash M:\tau$ is provable, with a proof having the same structure as the one of $C\,;u:\theta,\Gamma^\gamma \vdash M:\tau$.
(iii) If $C\,;u:\theta,\Gamma \vdash S$ and u does not occur in S then $C\,;\Gamma \vdash S$ is provable, with a proof having the same structure as the one of $C\,;u:\theta,\Gamma \vdash S$.
(iv) If $C\,;u:\theta,\Gamma \vdash [S\mid M]:\tau$ and u does not occur in S nor in M then $C\,;\Gamma \vdash [S\mid M]:\tau$ is provable, with a proof having the same structure as the one of $C\,;u:\theta,\Gamma \vdash [S\mid M]:\tau$.
(v) If $t::L,C\,;\Gamma^\gamma \vdash M:\tau$ and t does not occur free in Γ or τ then $C\,;\Gamma^\gamma \vdash M:\tau$ is provable, with a proof having the same structure as the one of $t::L,C\,;\Gamma^\gamma \vdash M:\tau$.

The proof, by induction on the inference of the typing judgements, is trivial. As usual regarding type safety, a crucial property to show is a "substitution lemma", relating typing and substitution. As a special case, we first show.

Lemma 4.5
$C\,;x:\sigma^\alpha,\Gamma^\gamma \vdash M:\tau \;\&\; y\notin \mathsf{dom}(\Gamma) \Rightarrow\; C\,;y:\sigma^\alpha,\Gamma^\gamma \vdash \{y\mapsto x\}M:\tau$.

Proof
By induction on the inference of $C\,;x:\sigma^\alpha,\Gamma^\gamma \vdash M:\tau$, trivial. □

Now we establish our crucial proposition.

Proposition 4.6
If Q is a type constraint such that $\mathsf{dom}(Q)\cap\mathsf{dom}(C_{\mathsf{typ}})=\emptyset$, and N is pure, then the following rule is admissible:

$$\frac{Q,C\,;\Gamma^\gamma \vdash N:\theta \qquad C\,;x:(\forall C'.\theta)^\alpha,\Gamma^\delta \vdash M:\tau}{C\,;\Gamma^\psi \vdash \{x\mapsto N\}M:\tau}$$

where

$$\psi=\begin{cases}\alpha_N\wedge\gamma\wedge\delta & \text{if } M \text{ is not pure}\\ \gamma\wedge\delta & \text{otherwise.}\end{cases}$$

Proof
First we observe that, if $x\notin\mathsf{fv}(M)$, then $\{x\mapsto N\}M=M$ and $C\,;\Gamma^\delta \vdash M:\tau$ by Lemma 4.4, and the proposition is easily established in this case, by using the degree weakening rule. Therefore we assume $x\in\mathsf{fv}(M)$ for the rest of the proof.

We proceed by induction on the inference of $C\,;x:(\forall Q.\theta)^\alpha,\Gamma^\delta \vdash M:\tau$, and by case on the last rule used to infer this sequent. This rule can only be either the degree

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

weakening rule, or a rule depending on the structure of M. In the first case, we simply use the induction hypothesis, and the degree weakening rule to conclude. Then we assume for the rest of the proof that the last rule is not the degree weakening rule; that is, we proceed by induction on the structure of M.

- If $M = x$, we have $C \vdash (\forall Q.\theta) \geq \tau$, that is $\tau = \mathsf{S}(\theta)$ for some $\mathsf{S} \in \mathscr{S}ub(Q, C)$, hence also $\mathsf{S} \in \mathscr{S}ub(C', C)$ where $C' = Q, C$. By Lemma 4.3 we have $C; \Gamma^{\gamma} \vdash N : \tau$, since S is the identity for the variables occurring in Γ, which are included into $\mathsf{dom}(C)$. We conclude using the degree weakening rule.
- If $M = \lambda z M'$ then we have $C ; x : (\forall Q.\theta)^{\beta}, z : \tau_0^a, \Gamma^{\kappa} \vdash M' : \tau_1$ for some β and κ, with $\alpha = 1$, $\delta = \lambda y.1$ and $\tau = \tau_0^a \to \tau_1$. We may assume that $z \notin \mathsf{dom}(\Gamma)$, hence $z \notin \mathsf{fv}(N)$. Then by Lemma 4.2 we also have $Q, C ; z : \tau_0^a, \Gamma^{\gamma} \vdash N : \theta$, and therefore $C ; (z : \tau_0, \Gamma)^{\xi} \vdash \{x \mapsto N\} M' : \tau_1$ by induction hypothesis, for some ξ such that $\xi(z) = a$. Then $C ; \Gamma^1 \vdash \{x \mapsto N\} M : \tau$ by the typing rule for abstraction, and we conclude using the degree weakening rule.
- If $M = (M_0 M_1)$ then there exist a, τ_0 and Σ^{κ} such that $C ; \Sigma^{\kappa} \vdash M_0 : \tau_0^a \to \tau$ and $C ; \Sigma^{\kappa} \vdash M_1 : \tau_0$, with $x : (\forall Q.\theta)^{\alpha}, \Gamma^{\delta} = \Sigma^{0_{M_0} \wedge \xi}$ where

$$\xi(y) = \begin{cases} a & \text{if } M_1 = y \\ (a_{M_1} \wedge \kappa)(y) & \text{otherwise} \end{cases}$$

We have $\Sigma^{\kappa} = x : (\forall Q.\theta)^{\beta}, \Gamma^{\kappa'}$ where κ' is the restriction of κ to $\mathsf{dom}(\Gamma)$ and $\beta = \kappa(x)$. By Lemmas 4.4(i) and 4.2(i), the judgement $C ; x : (\forall Q.\theta)^{\beta_0}, \Gamma^{\kappa'} \vdash M_0 : \tau_0^a \to \tau$ where $\beta_0 = \beta$ if $x \in \mathsf{fv}(M_0)$, and $\beta_0 = 1$ otherwise, is provable, with a proof having the same structure as the one of $C ; \Sigma^{\kappa} \vdash M_0 : \tau_0^a \to \tau$, and similarly for $C ; x : (\forall Q.\theta)^{\beta_1}, \Gamma^{\kappa'} \vdash M_1 : \tau_0$ with $\beta_1 = \beta$ if $x \in \mathsf{fv}(M_1)$, and $\beta_1 = 1$ otherwise. Then, by induction hypothesis $C ; \Gamma^{\psi_0} \vdash M_0' : \tau_0^a \to \tau$ and $C ; \Gamma^{\psi_1} \vdash M_1' : \tau_0$ where $M_i' = \{x \mapsto N\} M_i$ and

$$\psi_i = \begin{cases} \beta_{iN} \wedge \gamma \wedge \kappa' & \text{if } M_i \text{ is not pure} \\ \gamma \wedge \kappa' & \text{otherwise} \end{cases}$$

Then

$$C ; \Gamma^{0_{M_0'} \wedge \pi} \vdash \{x \mapsto N\} M : \tau$$

by the degree weakening rule and the rule for typing application, where

$$\pi(y) = \begin{cases} a & \text{if } M_1' = y \\ (a_{M_1'} \wedge \psi_0 \wedge \psi_1)(y) & \text{otherwise} \end{cases}$$

We notice that, since M is not pure, we have $\psi = \alpha_N \wedge \gamma \wedge \delta$. Then to conclude in this case, we have to check that

$$C \vdash (0_{M_0'} \wedge \pi)(y) \geqslant (\alpha_N \wedge \gamma \wedge \delta)(y) \quad \text{for} \quad y \in \mathsf{fv}(\{x \mapsto N\} M) \qquad (*)$$

We examine the two possible cases in the typing rule for application, that is M_1 is a variable (x or something else), or not.

1. If M_1 is a variable z, then $\psi_1 = \gamma \wedge \kappa'$ since M_1 is pure.

1.1. If $z = x$, we have $M_1' = N$. We again distinguish two cases:

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

1.1.1. If $x \notin \mathsf{fv}(M_0)$ then $M'_0 = M_0$, and $\beta_0 = 1$, therefore $C \vdash \psi_0 = \gamma \wedge \kappa'$. Moreover $C \vdash \alpha = a$ in this case. We have

$$C \vdash \pi \geqslant a_N \wedge \gamma \wedge \kappa'$$

and

$$C \vdash \delta(y) = (0_{M_0} \wedge \kappa')(y)$$

for $y \neq x$, hence the inequality $(*)$ is clearly true in this case.

1.1.2. If $x \in \mathsf{fv}(M_0)$ then $C \vdash \alpha = 0$, and therefore the inequality $(*)$ obviously holds for $y \in \mathsf{fv}(N)$. It also holds for $y \in \mathsf{fv}(M'_0) - \mathsf{fv}(N)$ since $C \vdash \delta(y) = 0$ in this case.

1.2. If $z \neq x$, we have $M'_1 = z$. Notice that, since we assumed $x \in \mathsf{fv}(M)$, we have $x \in \mathsf{fv}(M_0)$ in this case, hence $C \vdash \alpha = 0$. Therefore, the inequality $(*)$ obviously holds for $y \in \mathsf{fv}(N)$. It also holds for $y \in \mathsf{fv}(M_0) - \{x\}$, since $C \vdash \delta(y) = 0$ in this case. Finally, for $y \in \{z\} - (\mathsf{fv}(M_0) \cup \mathsf{fv}(N)$ we have $C \vdash (0_{M'_0} \wedge \pi)(y) = \pi(y) = a$ and $C \vdash \delta(y) = (0_{M_0} \wedge \xi)(y) = \xi(y) = a$, whence $(*)$ in this case.

2. Otherwise, M_1 is not a variable. Then $\xi = a_{M_1} \wedge \kappa$ and $\pi = a_{M'_1} \wedge \psi_0 \wedge \psi_1$. We have

$$C \vdash \alpha = \begin{cases} 0 & \text{if } x \in \mathsf{fv}(M_0) \\ a \wedge \beta & \text{otherwise} \end{cases}$$

since $x \in \mathsf{fv}(M)$. We distinguish again two cases:

2.1. If $x \in \mathsf{fv}(M_0)$ then $(*)$ is true for $y \in \mathsf{fv}(N)$. It also holds for $y \in \mathsf{fv}(M_0)$ since $C \vdash \delta(y) = 0$ in this case. Finally if $y \in \mathsf{fv}(M_1) - (\mathsf{fv}(M_0) \cup \mathsf{fv}(N))$ then $y \neq x$, and we have

$$\begin{aligned} C \vdash (0_{M'_0} \wedge \pi)(y) &= \pi(y) \\ &= a \wedge (\psi_0 \wedge \psi_1)(y) \\ &= a \wedge (\gamma \wedge \kappa')(y) && \text{for } C \vdash \psi_i(y) = (\gamma \wedge \kappa')(y) \\ &= (\gamma \wedge \delta)(y) && \text{for } C \vdash \delta(y) = \xi(y) = a \wedge \kappa'(y) \\ &= (\alpha_N \wedge \gamma \wedge \delta)(y) \end{aligned}$$

2.2. If $x \notin \mathsf{fv}(M_0)$ we have $M'_0 = M_0$ and $x \in \mathsf{fv}(M_1)$ (for we assumed that x is free in M). If $y \in \mathsf{fv}(M_0)$ then $C \vdash \delta(y) = 0$, and the inequality $(*)$ is true in this case. Otherwise, if $y \in \mathsf{fv}(N) - \mathsf{fv}(M_0)$, we have

$$\begin{aligned} C \vdash (0_{M'_0} \wedge \pi)(y) &= \pi(y) \\ &= a \wedge (\psi_0 \wedge \psi_1)(y) \\ &\geqslant a \wedge \beta \wedge (\gamma \wedge \kappa')(y) && \text{for } C \vdash \psi_i \geqslant \beta_N \wedge \gamma \wedge \kappa' \\ &\geqslant a \wedge \beta \wedge (\gamma \wedge \delta)(y) && \text{for } C \vdash \delta(y) = \xi(y) \leqslant \kappa'(y) \\ &= (\alpha_N \wedge \gamma \wedge \delta)(y) \end{aligned}$$

Finally, if $y \in \mathsf{fv}(M_1) - (\mathsf{fv}(M_0) \cup \mathsf{fv}(N))$ we conclude exactly as in the previous case.

- If $M = (\mathsf{let}\ z = M_0\ \mathsf{in}\ M_1)$ then $x : (\forall Q.\theta)^\alpha, \Gamma^\delta = \Sigma^\xi$ with $Q', C\ ;\ \Sigma^\kappa \vdash M_0 : \theta'$, where the constraint Q' is empty if M_0 is not pure, and $C\ ;\ z : (\forall Q'.\theta')^\beta, \Sigma^\kappa \vdash M_1 : \tau$

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

and

$$\xi = \begin{cases} \beta_{M_0} \wedge \kappa & \text{if } M_1 \text{ is not pure} \\ \kappa & \text{otherwise} \end{cases}$$

We may assume that $z \notin \mathsf{dom}(\Gamma)$ (and $z \neq x$), hence also $z \notin \mathsf{fv}(N)$, so that, by Lemma 4.2, $Q, C\,;\, z : (\forall Q'.\theta')^{\beta}, \Gamma^{\gamma} \vdash N : \theta$ and also $Q', Q, C\,;\, \Gamma^{\gamma} \vdash N : \theta$ by the same lemma. We have $\Sigma^{\kappa} = x : (\forall Q.\theta)^{\alpha'}, \Gamma^{\kappa'}$ where $\alpha' = \kappa(x)$ and κ' is the restriction of κ to $\mathsf{dom}(\Gamma)$. Then, by induction hypothesis

$$Q', C\,;\, \Gamma^{\psi_0} \vdash \{x \mapsto N\} M_0 : \theta'$$

and

$$C\,;\, z : (\forall Q'.\theta')^{\beta}, \Gamma^{\psi_1} \vdash \{x \mapsto N\} M_1 : \tau$$

where

$$\psi_i = \begin{cases} \alpha'_N \wedge \gamma \wedge \kappa' & \text{if } M_i \text{ is not pure} \\ \gamma \wedge \kappa' & \text{otherwise} \end{cases}$$

Let $M'_i = \{x \mapsto N\} M_i$. Since N is pure, by Lemma 2.2(i), if M'_0 is not pure, then M_0 is not pure. Then by the degree weakening rule, and the rule for the let construct, we have $C\,;\, \Gamma^{\pi} \vdash \{x \mapsto N\} M : \tau$ where

$$\pi = \begin{cases} \beta_{M'_0} \wedge \psi_0 \wedge \psi_1 & \text{if } M'_1 \text{ is not pure} \\ \psi_0 \wedge \psi_1 & \text{otherwise} \end{cases}$$

We distinguish two cases, according as to whether M'_1 is pure or not:

1. If M'_1 is not pure then, by the Lemma 2.2(i), M_1 is not pure (for N is pure), and therefore M is not pure. Then we have, observing that δ is the restriction of ξ to $\mathsf{dom}(\Gamma)$:

$$\begin{aligned} C \vdash \psi &= \alpha_N \wedge \gamma \wedge \delta \\ &= \alpha_N \wedge \gamma \wedge \beta_{M_0} \wedge \kappa' && \text{for } C \vdash \delta = \beta_{M_0} \wedge \kappa' \\ &= \left(\beta_{M_0}(x) \wedge \alpha'\right)_N \wedge \gamma \wedge \beta_{M_0} \wedge \kappa' && \text{for } C \vdash \alpha = \xi(x) = \beta_{M_0}(x) \wedge \alpha' \\ &= \beta_{M_0}(x)_N \wedge \beta_{M_0} \wedge \alpha'_N \wedge \gamma \wedge \kappa' \\ &= \beta_{M'_0} \wedge \alpha'_N \wedge \gamma \wedge \kappa' \end{aligned}$$

and

$$\begin{aligned} C \vdash \pi &= \beta_{M'_0} \wedge \psi_0 \wedge \psi_1 \\ &= \beta_{M'_0} \wedge \psi_0 \wedge \alpha'_N \wedge \gamma \wedge \kappa' \\ &= \beta_{M'_0} \wedge \alpha'_N \wedge \gamma \wedge \kappa' && \text{for } C \vdash \psi_0 \geqslant \alpha'_N \wedge \gamma \wedge \kappa' \\ &= \psi \end{aligned}$$

and we are done in this case.

2. If M'_1 is pure, then M_1 is pure, and therefore $\xi = \kappa$, hence $\alpha = \alpha'$ and $\delta = \kappa'$, and also $\psi_1 = \gamma \wedge \kappa'$ and $\pi = \psi_0 \wedge \psi_1$. There are two cases: if M_0 is pure, then M is pure, and therefore $\psi = \gamma \wedge \delta = \gamma \wedge \kappa'$. Then we have $C \vdash \pi = \gamma \wedge \kappa' = \psi$ since $\psi_0 = \psi_1$ in this case. If M_0 is not pure, then M is not pure, hence $\psi = \alpha_N \wedge \gamma \wedge \delta = \alpha_N \wedge \gamma \wedge \kappa'$. Since $\psi_0 = \alpha_N \wedge \gamma \wedge \kappa'$ in this case, we have $C \vdash \pi = \psi_0 \wedge \psi_1 = \psi$.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

- The case of (let rec $y = M_0$ in M_1) is similar, and all the other ones are easier. □

From the proof of this result, one can see that, apart from the cases where two typing contexts have to meet, there are only two occasions where we had to use the degree weakening rule: in cases 1.1.1 and 2.2 of $M = (M_0 M_1)$. The first one corresponds to $N = y$ and $M_1 = x$ with $x \notin \mathsf{fv}(M_0)$. Indeed, there are examples of terms of the form (let $x = y$ in $M_0 x$), which reduces into $\{x \mapsto y\}(M_0 x)$, for which y is safe for recursion, while the type systems says it is unsafe. For instance, the expression

$$(\mathsf{let\ rec}\ y = (\mathsf{let}\ x = y\ \mathsf{in}\ \mathsf{F}x)\ \mathsf{in}\ y)$$

where $\mathsf{F} = \lambda x y\, y$, is not accepted by the type system, because the degree of y in (let $x = y$ in Fx) is 0, while the reduced expression (let rec $y = \mathsf{F}y$ in y) is accepted. One could make a special case for N when it is a variable in the typing rule for the let construct (and have a similar distinction in the statement of Proposition 4.6), thus slightly enlarging the set of typable expressions. However, the proof would then be slightly more complicated, while we would not gain very much. In the other case (2.2), we have $x \notin \mathsf{fv}(M_0)$, M_1 is not a variable, and one (or both) of M_0 and M_1 is pure. Here the problem is that some variable of N may be downgraded, while it is safe for recursion, because x has degree 0 in M_1. For instance, if $M_0 = \mathsf{F}$, $M_1 = (\mathsf{let}\ x' = \mathsf{F}\ \mathsf{in}\ x)$ and $N = \lambda z\, y$, the expression

$$(\mathsf{let\ rec}\ y = (\mathsf{let}\ x = \lambda z y\ \mathsf{in}\ \mathsf{F}(\mathsf{let}\ x' = \mathsf{F}\ \mathsf{in}\ x))\ \mathsf{in}\ y)$$

which reduces to (let rec $y = \{x \mapsto N\}M$ in y), is rejected by the type system, while the latter one is accepted. However, it is unclear how one could improve the type system in this case.

To establish the "subject reduction" property, we also need a "replacement lemma" (see Wright & Felleisen (1994)) and a form of weakening involving generic instantiation of type schemes.

Lemma 4.7
Let M be an expression which is not pure. If $C\,;\Gamma^\gamma \vdash \mathbf{E}[M] : \tau$ is provable, with a sub-proof of $C'\,;\Delta^\delta \vdash M : \theta$ at the occurrence $\mathbf{E}$ of M, let N be such that $C'\,;\Delta^\delta \vdash N : \theta$. Then $C\,;\Gamma^\gamma \vdash \mathbf{E}[N] : \tau$.

Proof
By induction on the inference of $C\,;\Gamma^\gamma \vdash \mathbf{E}[M] : \tau$. In the case where $\mathbf{E} = (V\mathbf{E}')$ and $\mathbf{E}'[N]$ is a variable, we have to use the degree weakening rule, since $\mathbf{E}'[M]$ cannot be a variable. □

Lemma 4.8
If $C\,; x : \zeta^\alpha, \Gamma^\gamma \vdash M : \tau$ and $C \vdash \sigma \geq \zeta$ then $C\,; x : \sigma^\alpha, \Gamma^\gamma \vdash M : \tau$.

Proof
This is a standard result (see Damas & Milner (1982). □

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

Proposition 4.9 (Type Preservation)
(i) $C\,;\Gamma^{\gamma} \vdash M:\tau \;\&\; M \xrightarrow{*} N \;\Rightarrow\; C\,;\Gamma^{\gamma} \vdash N:\tau$,
(ii) if $C\,;\Gamma \vdash [S \mid M]:\tau$ with $u \in \mathsf{dom}(\Gamma) \;\Rightarrow\; u \in \mathsf{dom}(S)$ and $[S \mid M] \xrightarrow{*} [S' \mid N]$ then $C\,;\Delta \vdash [S \mid N]:\tau$ for some Δ.

Proof
(i) It is enough to prove this for $M \to N$. We proceed by induction on the proof of $M \to N$, and then by induction on the inference of $C\,;\Gamma^{\gamma} \vdash M:\tau$, and by case on the last rule used to infer this sequent. The case where this rule is the degree weakening is trivial, and is omitted.

- If $M = (\lambda x M')V$ and $N = \{x \mapsto V\}M'$, then the proof of $C\,;\Gamma^{\gamma} \vdash M:\tau$ has the following shape:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{C\,;x:\theta^{a},\Gamma^{\delta} \vdash M':\tau}}{C\,;\Gamma^{1} \vdash \lambda x M':\theta^{a}\to\tau}}{C\,;\Gamma^{\gamma} \vdash \lambda x M':\theta^{a}\to\tau} \qquad \dfrac{\vdots}{C\,;\Gamma^{\gamma} \vdash V:\theta}}{C\,;\Gamma^{0_{\lambda x M'}\wedge\xi} \vdash M:\tau}$$

with

$$\xi(y) = \begin{cases} a & \text{if } V = y \\ (a_V \wedge \gamma)(y) & \text{otherwise} \end{cases}$$

Then there are two cases:

1. If V is a variable z which does not occur free in $\lambda x M'$, then $\Gamma^{\gamma} = z:\sigma^{\alpha},\Delta^{\gamma'}$ with $C \vdash \sigma \geq \theta$. Then we also have $\Gamma^{\delta} = z:\sigma^{\beta},\Delta^{\delta'}$, hence $C\,;x:\theta^{a},\Delta^{\delta'} \vdash M':\tau$ by Lemma 4.4(i). Therefore $C\,;z:\theta^{a},\Delta^{\delta'} \vdash N:\tau$ by Lemma 4.5, hence $C\,;z:\sigma^{a},\Delta^{\delta'} \vdash N:\tau$ by Lemma 4.8. By Lemmas 4.4 and 4.2, we may assume that $\delta'(y) = 1$ for $y \notin \mathsf{fv}(M') - \{z\}$, and we are done in this case since $C \vdash 0_{\lambda x M'} \wedge \xi \leqslant \delta'$ on $\mathsf{fv}(\lambda x M')$.

2. Now assume that V is not a variable, or is a variable which occurs free in $\lambda x M'$. By the Proposition 4.6 we have $C\,;\Gamma^{\psi} \vdash N:\tau$ where

$$\psi = \begin{cases} a_V \wedge \gamma \wedge \delta & \text{if } M' \text{ is not pure} \\ \gamma \wedge \delta & \text{otherwise} \end{cases}$$

We obviously have $C \vdash \psi(y) \geqslant (0_{\lambda x M'} \wedge \xi)(y)$ for $y \in \mathsf{fv}(\lambda x M')$, while if $y \notin \mathsf{fv}(\lambda x M')$, we have $\xi(y) = (a_V \wedge \gamma)(y)$. In this case we may assume (using Lemmas 4.4 and 4.2 that $\delta(y) = 1$, and therefore $C \vdash \psi(y) \geqslant \xi(y)$. Then we conclude using the degree weakening rule.

- The case where $M = (\mathsf{let}\ x = V\ \mathsf{in}\ M')$ and $N = \{x \mapsto V\}M'$ is an immediate consequence of the Proposition 4.6. If $M = (\mathsf{let\ rec}\ x = V\ \mathsf{in}\ M')$ and $N =$

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

$\{x \mapsto (\mathsf{let\ rec}\ x = V\ \mathsf{in}\ V)\}M'$, we have

$$\frac{\dfrac{\vdots}{C', C\ ;\ x : \theta^1, \Gamma^\gamma \vdash V : \theta} \qquad \dfrac{\vdots}{C\ ;\ x : (\forall C'.\theta)^\alpha, \Gamma^\gamma \vdash M' : \tau}}{C\ ;\ \Gamma^\delta \vdash (\mathsf{let\ rec}\ x = V\ \mathsf{in}\ M') : \tau}$$

where

$$\delta \ = \ \begin{cases} \alpha_V \wedge \gamma & \text{if } M' \text{ is not pure} \\ \gamma & \text{otherwise} \end{cases}$$

and therefore

$$\frac{\dfrac{\vdots}{C', C\ ;\ x : \theta^1, \Gamma^\gamma \vdash V : \theta} \qquad \dfrac{\vdots}{C', C\ ;\ x : \theta^1, \Gamma^\gamma \vdash V : \theta}}{C', C\ ;\ \Gamma^\gamma \vdash (\mathsf{let\ rec}\ x = V\ \mathsf{in}\ V) : \theta}$$

by the rule for the let rec construct. Then $C\ ;\ \Gamma^\delta \vdash N : \tau$ by the Proposition 4.6.

All the other cases are straightforward. For the reductions $\mathbf{E}[M] \to \mathbf{E}[M']$ with $M \to M'$, we proceed by induction on $\mathbf{E}$, using the induction hypothesis. In the case where $\mathbf{E} = (V\mathbf{E}')$ and $\mathbf{E}'[M']$ is a variable, we have to use the degree weakening rule, since $\mathbf{E}'[M]$ cannot be a variable. In the case where $\mathbf{E} = (\mathsf{let}\ x = \mathbf{E}'\ \mathsf{in}\ N)$ or $\mathbf{E} = (\mathsf{let\ rec}\ x = \mathbf{E}'\ \mathsf{in}\ N)$, and $\mathbf{E}'[M]$ is pure, we use the Lemma 2.2(ii).

(ii) Again it is enough to prove this for $[S \mid M] \to [S' \mid N]$. We proceed by induction on the proof of this reduction. We have $C\ ;\ \Gamma \vdash S$ and $C\ ;\ \Gamma^\gamma \vdash M : \tau$. If $M \to N$, then we use the previous point. If $M = \mathbf{E}[(\mathsf{ref}\ V)]$ with $S' = (u := V\ ; S)$, $N = \mathbf{E}[u]$ and u does not occur in S or $\mathbf{E}[V]$ (and $\mathsf{fv}(V) \cap \mathsf{capt}(\mathbf{E}) = \emptyset$), then we may assume, by Lemma 4.4, that $u \notin \mathsf{dom}(\Gamma)$. In the proof of $C\ ;\ \Gamma^\gamma \vdash M : \tau$ there is a sub-proof regarding $(\mathsf{ref}\ V)$, of the form

$$\cfrac{\cfrac{\cfrac{\cfrac{\vdots}{C \vdash \Sigma}}{C\ ;\ \Sigma^1 \vdash \mathsf{ref} : \theta^0 \to \theta\ \mathsf{ref}} \qquad \cfrac{\vdots}{C\ ;\ \Sigma^\xi \vdash V : \theta}}{C\ ;\ \Sigma^\kappa \vdash (\mathsf{ref}\ V) : \theta\ \mathsf{ref}}}{C\ ;\ \Sigma^\delta \vdash (\mathsf{ref}\ V) : \theta\ \mathsf{ref}}$$

with $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Sigma) - \mathsf{capt}(\mathbf{E})$. Since $u \notin \mathsf{dom}(\Gamma)$, by weakening there is a proof of $C\ ;\ u : \theta, \Gamma^\gamma \vdash M : \tau$ having the same structure as the proof of $C\ ;\ \Gamma^\gamma \vdash M : \tau$. In particular, we have a sub-proof of $C ; u : \theta, \Sigma^\delta \vdash (\mathsf{ref}\ V) : \theta\ \mathsf{ref}$ with $C ; u : \theta, \Sigma^\xi \vdash V : \theta$,

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

and therefore $C; u : \theta, \Sigma \vdash S'$. Obviously, $C; u : \theta, \Sigma^{\xi} \vdash u : \theta$, hence $C; u : \theta, \Gamma^{\gamma} \vdash \mathbf{E}[u] : \tau$ by Lemma 4.7, and we let $\Delta = u : \theta, \Sigma$ in this case. The other cases of global reduction are similar. □

An immediate corollary of this result and of the Corollary 3.3 is that one never needs the value of a recursive variable while evaluating the body of a typable recursive definition.

Corollary 4.10
If (let rec $x = N$ in M) is typable and $N \xrightarrow{*} \mathbf{E}[x]$, then $\mathbf{E} = \mathbf{E}'[(V\ \square)]$, where V is a protective function.

This result is exploited in Boudol & Zimmer (2002) to design a provably correct abstract machine for call-by-value recursion. Now combining the type preservation property with Corollary 2.7 and Lemma 4.1, we get:

Theorem 4.11 (Type Safety)
For any closed configuration $[S \mid M]$, if it is typable, i.e. $C ; \Gamma \vdash [S \mid M] : \tau$ for some C, Γ and τ, and if its evaluation terminates on $[S' \mid N]$, then N is a value (and not a faulty expression) of type τ.

5 Type assignment

In this section we prove that the standard result about typability in ML, namely that one can compute a principal type for any typable program, extends to our language. Regarding the record calculus, this was established by Jategaonkar & Mitchell (1993), though with a different view of row variables. Then the main novelty here is that we deal with syntactically unrestricted recursion, and a type system which includes a new ingredient, the degrees. Nevertheless, since our approach regarding records is slightly different from that of Jategaonkar & Mitchell (1993), we present the type assignment algorithm in full detail.

The data for the type assignment algorithm are a term M, a constraint C, and a type assumption Γ for some variables, and the algorithm, if it does not fail, yields a type τ, a degree assignment γ, a constraint C' and a substitution $\mathsf{S} \in \mathscr{S}ub(C, C')$ such that $C' ; \mathsf{S}(\Gamma)^{\gamma} \vdash M : \tau$ is a valid typing. As usual, type assignment involves solving *verification conditions*. Here we have not only to solve type equations, by means of unification, but also degree inequalities. Then our verification conditions are of the form $(\mathscr{Q}.\ A ; E)$ where

- $\mathscr{Q}$ is a sequence of existential quantifications $\exists t$ over type variables, the scope of which is $A ; E$;
- A is a set of annotation assertions $\sigma :: L$ and of degree inequalities of the form $a \leqslant \alpha$;
- E is a set of type equations $\tau = \theta$ (to be solved), and of degree equations, of the form $a = b$ or $\alpha = 1$.

Notice that the degree equations that we have to deal with are quite trivial to solve (recall that a and b are either constants, 0 or 1, or variables, but do not involve the

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

$\wedge$ operation). We shall only deal with verification conditions $(\mathcal{Q}.\ A\,;E)$ satisfying the requirement that the type variables occurring in the equations of E also occur in the annotation assertions of A. We use existential quantification to deal with the "fresh variables" that the algorithm may introduce (see Jouannaud & Kirchner (1991)). Let us denote by $\mathsf{fv}(E)$, and similarly $\mathsf{var}(\mathcal{Q})$, the set of (type and degree) variables occurring in E (resp. in $\mathcal{Q}$).

Definition 5.1
A *solution* of the verification condition $(\mathcal{Q}.\ A\,;E)$ is a pair (C,S) such that there exists S' with $\mathsf{S} = \mathsf{S}' \restriction(\mathsf{fv}(E) - \mathsf{var}(\mathcal{Q}))$ and

(i) $C \vdash \mathsf{S}'(A)$;
(ii) $\tau = \tau' \in E \ \Rightarrow\ C \vdash \mathsf{S}'(\tau) :: \emptyset,\ C \vdash \mathsf{S}'(\tau') :: \emptyset$ and $\mathsf{S}'(\tau) =_{\mathscr{T}} \mathsf{S}'(\tau')$;
(iii) $\alpha = \beta \in E \ \Rightarrow \vdash \mathsf{S}'(\alpha) = \mathsf{S}'(\beta)$.

A solution (C,S) is more general than (C',S') if there exists a substitution $\mathsf{S}'' \in \mathscr{S}ub(C,C')$ such that $\mathsf{S}' = \mathsf{S}''\mathsf{S}$.

As one can see, if (C,S) is a solution of $(\mathcal{Q}.\ A\,;E)$, then S is the identity for the variables that do not occur free in this verification condition. As usual (see Martelli & Montanari (1982) and Jouannaud & Kirchner (1991)), solving $(\mathcal{Q}.\ A\,;E)$ consists in transforming it into a "solved form".

Definition 5.2
A verification condition $(\mathcal{Q}.\ A\,;E)$ is a *solved form* if A is a constraint in the sense of the type system, that is $A = \{\, t_h :: L_h \mid h \in H \,\} \cup \{\, p_k \leqslant \alpha_k \mid k \in K \,\}$, and $E = \{\, t'_i = \tau_i \mid i \in I \,\} \cup \{\, q_j = a_j \mid j \in J \,\}$ with:

(i) the type variables t'_i only occur in E as the left members of the equations $t'_i = \tau_i$;
(ii) the degree variables q_j only occur in A and E as the left members of the equations $q_j = a_j$;
(iii) $\{\, t'_i \mid i \in I \,\} \subseteq \{\, t_h \mid h \in H \,\}$ and $A \vdash \tau_i :: A(t'_i)$ for all i.

We shall use the notation $(\mathcal{Q}.\ C\,;S)$ for solved forms. Such a solved form has an obvious solution, namely (C,S) where $\mathsf{S} = \mathsf{S}' \restriction(\mathsf{fv}(S) - \mathsf{var}(\mathcal{Q}))$, with $\mathsf{S}' = \{\, t'_i \mapsto \tau_i \mid i \in I \,\} \cup \{\, q_j \mapsto a_j \mid j \in J \,\}$. We can check that $\mathsf{S} \in \mathscr{S}ub(C_0, C)$ where $C_0 = \{\, t'_i :: \emptyset \mid i \in I \,\}$, using the following lemma.

Lemma 5.3
$C \vdash \sigma :: L \ \&\ L' \subseteq L \ \Rightarrow\ C \vdash \sigma :: L'$.

(The proof, by induction on the inference of $C \vdash \sigma :: L$, is trivial.)

It is easy to see that the "canonical solution" S – as just described – of a solved form $(\mathcal{Q}.\ C\,;S)$ is also a most general solution, because any other solution S' is such that $\mathsf{S}' = \mathsf{S}''\mathsf{S}$ where $\mathsf{S}''(t) = \mathsf{S}'(t)$ except for $t \in \{\, t'_i \mid i \in I \,\}$ and similarly for $\mathsf{S}'(p)$. To show that S'' is indeed an acceptable substitution, we need to use the following lemma.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

$$t :: L_0, t :: L_1, A \quad \rhd \quad t :: L_0 \cup L_1, A \tag{1}$$

$$\mathsf{unit} :: \emptyset, A \quad \rhd \quad A \tag{2}$$

$$(\theta^a \to \tau) :: \emptyset, A \quad \rhd \quad \theta :: \emptyset, \tau :: \emptyset, A \tag{3}$$

$$\tau\,\mathsf{ref} :: \emptyset, A \quad \rhd \quad \tau :: \emptyset, A \tag{4}$$

$$\langle\rangle :: L, A \quad \rhd \quad A \tag{5}$$

$$\langle \rho, \ell : \tau \rangle :: L, A \quad \rhd \quad \rho :: (L \cup \{\ell\}), \tau :: \emptyset, A \qquad (\ell \notin L) \tag{6}$$

$$\frac{\sigma :: \emptyset \rhd^* C}{(\forall t :: L.\sigma) :: \emptyset, A \rhd C \backslash t, A} \; C(t) \subseteq L,\ t \notin A \tag{7}$$

Fig. 8. Decomposition of annotation assertions.

Lemma 5.4
If $C \vdash \mathsf{S}(\tau) :: L$ then there exists C' such that $\mathsf{dom}(C') = \mathsf{fv}(\tau)$ with $C' \vdash \tau :: L$ and $C \vdash \mathsf{S}(t) :: C'(t)$ for any $t \in \mathsf{fv}(\tau)$.

(Again, the proof is straightforward.)

Now we define the transformations of verification conditions. First, we have a set of transformations to decompose – if this does not fail – a set of annotation assertions A into a constraint C. These transformations $A \rhd A'$ are described in figure 8, where we do not include the rule that decomposing annotation assertions holds up to α-conversion of type schemes. In this figure, $C\backslash t$ denotes the restriction of C to variables different from t, that is $C\backslash t = C \restriction(\mathsf{dom}(C) - \{t\})$. In the last rule, $\rhd^*$ is the reflexive and transitive closure of $\rhd$. The following lemma asserts the correctness of this decomposition process.

Lemma 5.5
$C \vdash A$ if and only if there exists C' such that $A \rhd^* C'$ and $C \vdash C'$.

Proof
For the $\Leftarrow$ direction, we prove that $A \rhd^* A'$ and $C \vdash A'$ implies $C \vdash A$, by induction on the definition of $A \rhd^* A'$. Conversely, we show that $C \vdash \sigma :: L$ implies $\sigma :: L \rhd^* C'$ for some $C' \subseteq C$ (which means that $\mathsf{dom}(C') \subseteq \mathsf{dom}(C)$ and $C'(t) \subseteq C(t)$ for any t). We conclude using the fact that if $C \vdash A$ and $C' \vdash A'$ then $C \cup C' \vdash A, A'$ where $C \cup C'$ is the constraint given by $\mathsf{dom}(C \cup C') = \mathsf{dom}(C) \cup \mathsf{dom}(C')$, and $(C \cup C')(t) = C(t) \cup C'(t)$, where by convention $C(t) = \emptyset$ if $t \notin \mathsf{dom}(C)$. □

Lemma 5.6
For any substitution S, if $A \rhd A'$ and $C \vdash \mathsf{S}(A)$ then $C \vdash \mathsf{S}(A')$.

The proof, by induction on $A \rhd A'$, is easy (one uses the fact that if $C \vdash \sigma :: L$ and $C \vdash \sigma :: L'$ then $C \vdash \sigma :: L \cup L'$).

Now we show how to decompose verification conditions – we still use the notation $\rhd$. As usual (see Jouannaud & Kirchner (1991)), the intention is that this decomposition either fails, meaning that the verification condition has no solution, or terminate on a solved form, while preserving the set of solutions. The decomposition relation, including decomposition of annotation assertions and of

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

$$\frac{A \rhd A'}{(\mathcal{Q}.A\,;E) \rhd (\mathcal{Q}.A'\,;E)} \tag{8}$$

$$(\mathcal{Q}.\ p \leqslant \alpha, p \leqslant \beta, A\,;E) \rhd (\mathcal{Q}.\ p \leqslant \alpha \wedge \beta, A\,;E) \tag{9}$$

$$(\mathcal{Q}.\ 0 \leqslant \alpha, A\,;E) \rhd (\mathcal{Q}.\ A\,;E) \tag{10}$$

$$(\mathcal{Q}.\ 1 \leqslant \alpha, A\,;E) \rhd (\mathcal{Q}.\ A\,;\alpha = 1, E) \tag{11}$$

$$(\mathcal{Q}.\ A\,;(\alpha \wedge \beta) = 1, E) \rhd (\mathcal{Q}.\ A\,;\alpha = 1, \beta = 1, E) \tag{12}$$

$$(\mathcal{Q}.\ A\,;q = a, E) \rhd (\mathcal{Q}.\ \{q \mapsto a\}A\,;q = a, \{q \mapsto a\}E) \tag{13}$$

where $q \in (\mathsf{fv}(A) \cup \mathsf{fv}(E)) - \{a\}$

$$(\mathcal{Q}.\ A\,;\mathbf{e} = \mathbf{e}, E) \rhd (\mathcal{Q}.\ A\,;E) \tag{14}$$

$$(\mathcal{Q}.\ A\,;\mathbf{e} = \mathbf{x}, E) \rhd (\mathcal{Q}.\ A\,;\mathbf{x} = \mathbf{e}, E) \qquad (\mathbf{e} \notin \mathcal{V}ar,\ \mathbf{x} \in \mathcal{V}ar) \tag{15}$$

$$(\mathcal{Q}.\ A\,;\theta_0^a \to \tau_0 = \theta_1^b \to \tau_1, E) \rhd (\mathcal{Q}.\ A\,;E', E) \tag{16}$$

where $E' = \{\theta_0 = \theta_1, \tau_0 = \tau_1, a = b\}$

$$(\mathcal{Q}.\ A\,;\tau_0\ \mathsf{ref} = \tau_1\ \mathsf{ref}, E) \rhd (\mathcal{Q}.\ A\,;\tau_0 = \tau_1, E) \tag{17}$$

$$(\mathcal{Q}.\ A\,;\langle \rho_0, \ell : \tau_0\rangle = \langle \rho_1, \ell : \tau_1\rangle, E) \rhd (\mathcal{Q}.\ A\,;\rho_0 = \rho_1, \tau_0 = \tau_1, E) \tag{18}$$

$$(\mathcal{Q}.\ A\,;\langle \rho_0, \ell_0 : \tau_0\rangle = \langle \rho_1, \ell_1 : \tau_1\rangle, E) \rhd (\mathcal{Q}.\exists r.\ A'\,;E', E) \tag{19}$$

where $A' = r :: \{\ell_0, \ell_1\}, A$

$E' = \{\rho_0 = \langle r, \ell_1 : \tau_1\rangle, \rho_1 = \langle r, \ell_0 : \tau_0\rangle\}$

if $\ell_0 \neq \ell_1$, where r is fresh.

$$(\mathcal{Q}.\ A\,;t = \tau, E) \rhd (\mathcal{Q}.\ A\,;t = \tau, \{t \mapsto \tau\}E) \tag{20}$$

$t \in \mathsf{fv}(E) - \mathsf{fv}(\tau)$

$$\frac{\{\tau :: C(t) \mid t = \tau \in S\}, C \rhd^* C'}{(\mathcal{Q}.\ C\,;S) \rhd (\mathcal{Q}.\ C'\,;S)}\ \{t = \tau \mid t = \tau \in S\ \&\ C \nvdash \tau :: C(t)\} \neq \emptyset \tag{21}$$

Fig. 9. Decomposition of verification conditions.

type equations, is described in figure 9, where we use some loose notations (such as $\tau = \tau', E$ for $\{\tau = \tau'\} \cup E$) and a notion of "fresh" variable, meaning "not occurring in the current context". In clauses (14) and (15) we use the symbols **e** and **x** to denote respectively either a type or degree expression, and either a type or degree variable. In the last rule S denotes a set of equations satisfying the clause (i) of Definition 5.2.

Proposition 5.7

(i) The decomposition $\rhd$ terminates.
(ii) If $(\mathcal{Q}.\ A\,;E) \rhd (\mathcal{Q}'.\ A'\,;E')$ then the variables occurring free in $(\mathcal{Q}'.\ A'\,;E')$ also occur in $(\mathcal{Q}.\ A\,;E)$, and $(\mathcal{Q}.\ A\,;E)$ and $(\mathcal{Q}'.\ A'\,;E')$ have the same solutions.
(iii) If $(\mathcal{Q}.\ A\,;E)$ is irreducible with respect to $\rhd$, then it has a solution if and only if it is a solved form.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

Proof

For any type scheme σ, let $\#(\sigma)$ be the number of logical symbols occurring in σ. Then, for instance, $\#(\forall t :: L.\sigma) = \#(\sigma) + 1$, and so on. For any set $A = \{ \sigma_h :: L_h \mid h \in H \} \cup \{ a_k \leqslant \alpha_k \mid k \in K \}$ of annotation assertions and degree inequations, we define its size $|A|$ to be (m, n) where $m = \sum\{ \#(\sigma_h) \mid h \in H \}$ and n is the number of elements of H. Let $\leqslant$ be the lexicographic ordering on tuples of integers (of a given length). Then we have:

$$A \rhd A' \;\Rightarrow\; |A| > |A'|$$

This is easy to see, by induction on the definition of $A \rhd A'$: rules (1, 2, 5) strictly decrease the number n of annotation assertions, and do not increase the sum m of the size of the type schemes, while rules (3, 4, 6) – and also (2, 5) – strictly decrease the sum m of the size of the type schemes. Finally, for rule (7) we have $|\sigma :: \emptyset| \geqslant |C|$ by induction, and therefore $|(\forall t :: L.\sigma) :: \emptyset, A| > |C \backslash t, A|$.

Now, for any type equation $\tau = \tau'$, let $\#(\tau = \tau') = \#(\tau) + \#(\tau')$, and similarly $\#(\alpha = \beta) = \#(\alpha) + \#(\beta) = \#(\alpha \leqslant \beta)$ where the size of a degree expression is the number of symbols occurring in it. Then we define the size $|(\mathcal{Q}.\ A\ ;\ E)|_k$ of a verification condition with respect to some integer k as follows:

$$|(\mathcal{Q}.A\ ;\ E)|_k = (n, n', n_k, \ldots, n_0, m, l, |A|)$$

where:

- n is the number of degree inequalities in A, that is the number of elements of K,
- n' is the number of type variables which do not occur only once as the left-hand side of some equation of E, but may occur in A (in particular, this number is strictly positive if E contains an equation $\tau = t$, or $t = \tau, E'$ and t occurs elsewhere), plus the number of degree variables which do not occur only once as the left-hand side of some equation of E,
- n_i is the number of inequations of size i in A, plus the number of equations of size i in E,
- m is the number of equations in E of the form $\mathbf{e} = \mathbf{x}$ with $\mathbf{e} \notin \mathcal{V}ar$ and $\mathbf{x} \in \mathcal{V}ar$, and
- l is 0 if (A, E) is not a (C, S), and the number of equations $t = \tau$ of S such that $C \not\vdash \tau :: C(t)$ otherwise.

It is easy to see that

$$\begin{aligned}(\mathcal{Q}.\ A\ ;\ E) \rhd (\mathcal{Q}'.\ A'\ ;\ E') \;\Rightarrow\; &\mathsf{max}\{ \#(\mathbf{e} = \mathbf{e}') \mid \mathbf{e} = \mathbf{e}' \in E \} \\ &\geqslant \mathsf{max}\{ \#(\mathbf{e} = \mathbf{e}') \mid \mathbf{e} = \mathbf{e}' \in E' \}\end{aligned}$$

Now let k be such that $k \geqslant \mathsf{max}\{ \#(\mathbf{e} = \mathbf{e}') \mid \mathbf{e} = \mathbf{e}' \in E \}$. We prove that

$$(\mathcal{Q}.\ A\ ;\ E) \rhd (\mathcal{Q}'.\ A'\ ;\ E') \;\Rightarrow\; |(\mathcal{Q}.\ A\ ;\ E)|_k \;>\; |(\mathcal{Q}'.\ A'\ ;\ E')|_k$$

by induction on the definition of $\rhd$. The rule (8) strictly decreases the size of the set of annotation assertions, as we have seen, while not affecting the other components. Rules (9), (10) and (11) strictly decrease n, that is the number of degree inequalities

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

in A. Rules (12, 14, 16, 17, 18, 19) strictly decrease some n_i (notice that (19) does not decrease the sum of the size of the type equations). Rule (15) strictly decreases the number m of equations of the form $\tau = t$ (and possibly also n), and rules (13) and (20) strictly decrease n'. Finally, let us see that (21) strictly decreases l: since for any equation $t = \tau$ of S the variable t does not occur in the right-hand side of an equation of S, decomposing $\{\, \tau :: C(t) \mid t = \tau \in S \,\}, C$ into C' cannot add any new assertion about t, that is $C'(t) = C(t)$. By Lemma 5.5, we then have $C' \vdash \tau :: C'(t)$ for any $t = \tau \in S$. This concludes the proof of the point (i) of the proposition.

The point (ii) is almost trivial. Let us just see the case of rule (19). Let $\mathsf{S} \in \mathscr{S}ub(C_0, C_1)$ be a solution of $(\mathcal{2}.\ A\ ;\ \langle \rho_0, \ell_0 : \tau_0 \rangle = \langle \rho_1, \ell_1 : \tau_1 \rangle, E)$, and let C'_0 and S' be as in Definition 5.1. Then $r \notin \mathsf{dom}(C'_0)$ since r is fresh. Let $C''_0 = r :: \{\ell_0, \ell_1\}, C'_0$. Since $\mathsf{S}'\langle \rho_0, \ell_0 : \tau_0 \rangle =_{\mathscr{T}} \mathsf{S}'\langle \rho_1, \ell_1 : \tau_1 \rangle$ we have $\mathsf{S}'(\rho_0) =_{\mathscr{T}} \langle \rho, \ell_1 : \mathsf{S}'(\tau_1) \rangle$ and $\mathsf{S}'(\rho_1) =_{\mathscr{T}} \langle \rho, \ell_0 : \mathsf{S}'(\tau_0) \rangle$. Let $\mathsf{S}'' = \mathsf{S}' \cup \{r \mapsto \rho\}$. Since $C_1 \vdash \mathsf{S}'\langle \rho_0, \ell_0 : \tau_0 \rangle :: \emptyset$ and $C_1 \vdash \mathsf{S}'\langle \rho_1, \ell_1 : \tau_1 \rangle :: \emptyset$, we have $C_1 \vdash \rho :: \{\ell_0, \ell_1\}$, and therefore $\mathsf{S}'' \in \mathscr{S}ub(C''_0, C_1)$, and $C_1 \vdash \mathsf{S}''(A')$ where $A' = r :: \{\ell_0, \ell_1\}, A$. It is easy to see that the other conditions for $\mathsf{S} \in \mathscr{S}ub(C_0, C_1)$ to be a solution of $(\mathcal{2}.\exists r.\ A'\ ;\ E', E)$ are met, where $E' = \{\rho_0 = \langle r, \ell_1 : \tau_1 \rangle, \rho_1 = \langle r, \ell_0 : \tau_0 \rangle\}$. Conversely, if $\mathsf{S} \in \mathscr{S}ub(C_0, C_1)$ is a solution of $(\mathcal{2}.\exists r.\ A'\ ;\ E', E)$, we have to check that $C_1 \vdash \mathsf{S}'\langle \rho_0, \ell_0 : \tau_0 \rangle :: \emptyset$ (again using S' as given in Definition 5.1), and similarly for $\langle \rho_1, \ell_1 : \tau_1 \rangle$. We have

$$\begin{aligned} \mathsf{S}'\langle \rho_0, \ell_0 : \tau_0 \rangle &=_{\mathscr{T}} \langle \mathsf{S}'(\rho_0), \ell_0 : \mathsf{S}'(\tau_0) \rangle \\ &=_{\mathscr{T}} \langle \langle \mathsf{S}'(r), \ell_1 : \mathsf{S}'(\tau_1) \rangle, \ell_0 : \mathsf{S}'(\tau_0) \rangle \end{aligned}$$

Since $C_1 \vdash \mathsf{S}'(\tau_i) :: \emptyset$ and $C_1 \vdash \mathsf{S}'(r) :: \{\ell_0, \ell_1\}$, we easily conclude.

Regarding the last point, we have seen that solved forms are solvable. Now if $(\mathcal{2}.\ A\ ;\ E)$ is irreducible with respect to $\rhd$, but is not a solved form, then a case analysis shows that $(\mathcal{2}.\ A\ ;\ E)$ has no solution. This holds in particular if E contains $0 = 1$ or $1 = 0$. □

An immediate corollary is as follows.

Corollary 5.8
If a verification condition is solvable, then it has a most general solution.

We denote by $\mathscr{S}ol(\mathcal{2}.\ A\ ;\ E)$ the most general solution of the verification condition $(\mathcal{2}.\ A\ ;\ E)$, if it exists (in which case it is unique, up to a permutation of the variables).

Now, coming back to the issue of type inference, we describe the algorithm for type assignment as a function $\mathsf{Type}(C, \Gamma, M) = (\tau, \gamma, (C', \mathsf{S}))$, with the idea that $\mathsf{S} \in \mathscr{S}ub(C, C')$ and $C'\ ;\ \mathsf{S}(\Gamma)^{\gamma} \vdash M : \tau$. However, this will be true only if $C \vdash \Gamma$. We could report a failure otherwise, but we shall actually use the algorithm only in this case. We assume here that M does not contain any location, although the algorithm could easily be extended to cover this case too, and that Γ assigns some type to each free variable of M (we should otherwise report a failure). The function Type is defined up to α-conversion performed in Γ and M, and the type and degree variables introduced by the algorithm are implicitly assumed to be fresh. In the definition of the algorithm, we use the notation $\mathsf{S}\backslash t$ for the substitution that coincides with S, except for $(\mathsf{S}\backslash t)(t) = t$. We also abusively write $\mathsf{S}\backslash C$ for the substitution that

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

coincides with S, except on $\mathsf{dom}(C)$, where it is the identity. We denote by $C \restriction \Gamma$ the constraint defined as follows: $(C \restriction \Gamma)_{\mathsf{deg}} = C_{\mathsf{deg}}$ and $(C \restriction \Gamma)_{\mathsf{typ}} = C_{\mathsf{typ}} \restriction \mathsf{fv}(\Gamma)$. When we write $(C', \mathsf{S}) = \mathscr{S}ol(\mathscr{Q}.\ A\,;E)$ in the definition of $\mathsf{Type}(C, \Gamma, M)$, we mean that the algorithm reports a failure (treated as an exception) in the case where $\mathscr{S}ol(\mathscr{Q}.\ A\,;E)$ does not exist, and similarly when $A \rhd^* C'$ is used.

- If $\Gamma(x) = (\forall C_0.\tau)$ with $\mathsf{dom}(C_0) \cap \mathsf{dom}(C) = \emptyset$,
  then $\mathsf{Type}(C, \Gamma, x) = (\tau, 0_x, (C \cup C_0, \mathsf{id}))$.
- If $\mathsf{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Gamma, M) = (\tau, \gamma, (C', \mathsf{S}))$
  then $\mathsf{Type}(C, \Gamma, \lambda x M) = (\mathsf{S}(t)^p \to \tau, 1, (\{p \leqslant \gamma(x)\} \cup C', \mathsf{S}\backslash t))$.
- If $\mathsf{Type}(C, \Gamma, M) = (\tau_0, \gamma_0, (C_0, \mathsf{S}_0))$
  and $\mathsf{Type}(C_0', \mathsf{S}_0(\Gamma), N) = (\tau_1, \gamma_1, (C_1, \mathsf{S}_1))$ where $C_0' = C_0 \restriction \mathsf{S}_0(\Gamma)$
  and $(C', \mathsf{S}) = \mathscr{S}ol(\{t :: \emptyset\} \cup C_1\,;\{\mathsf{S}_1(\tau_0) = \tau_1^p \to t\})$
  then $\mathsf{Type}(C, \Gamma, (MN)) = (\mathsf{S}(t), 0_M \wedge \mathsf{S}(\delta), (C', \mathsf{S}\mathsf{S}_1\mathsf{S}_0))$ where
$$\delta(x) \;=\; \begin{cases} p & \text{if } N = x \\ p \wedge \gamma_1(x) & \text{otherwise} \end{cases}$$
- If $\mathsf{Type}(C, \Gamma, N) = (\tau_0, \gamma_0, (C_0, \mathsf{S}_0))$
  and C_2 is $C_0\backslash\mathsf{fv}(\mathsf{S}_0(\Gamma))$, if N is pure, and $C_2 = \emptyset$ otherwise
  and $\mathsf{Type}(C_0 - C_2, \{x : (\forall C_2.\tau_0)\} \cup \mathsf{S}_0(\Gamma), M) = (\tau_1, \gamma_1, (C_1, \mathsf{S}_1))$
  then $\mathsf{Type}(C, \Gamma, (\mathsf{let}\ x = N\ \mathsf{in}\ M)) = (\tau_1, \delta, (C_1, \mathsf{S}_1\mathsf{S}_0))$ where
$$\delta \;=\; \begin{cases} \gamma_1(x)_N \wedge \mathsf{S}_1(\gamma_0) \wedge \gamma_1 & \text{if } M \text{ is not pure} \\ \mathsf{S}_1(\gamma_0) \wedge \gamma_1 & \text{otherwise} \end{cases}$$
- If $\mathsf{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Gamma, N) = (\tau_0, \gamma_0, (C_0, \mathsf{S}_0))$
  and $(C_1, \mathsf{S}_1) = \mathscr{S}ol(C_0\,;\{\mathsf{S}_0(t) = \tau_0, \gamma_0(x) = 1\})$
  and C_2 is $C_1\backslash\mathsf{fv}(\mathsf{S}_1\mathsf{S}_0(\Gamma))$, if N is pure, and $C_2 = \emptyset$ otherwise
  and $\mathsf{Type}(C_1 - C_2, \{x : (\forall C_2.\mathsf{S}_1\mathsf{S}_0(t))\} \cup \mathsf{S}_1\mathsf{S}_0(\Gamma), M) = (\tau, \gamma_1, (C', \mathsf{S}))$
  then $\mathsf{Type}(C, \Gamma, (\mathsf{let\ rec}\ x = N\ \mathsf{in}\ M)) = (\tau, \delta, (C', \mathsf{S}\mathsf{S}_1\mathsf{S}_0))$ where
$$\delta \;=\; \begin{cases} \gamma_1(x)_N \wedge \mathsf{S}\mathsf{S}_1(\gamma_0) \wedge \gamma_1 & \text{if } M \text{ is not pure} \\ \mathsf{S}\mathsf{S}_1(\gamma_0) \wedge \gamma_1 & \text{otherwise} \end{cases}$$
- $\mathsf{Type}(C, \Gamma, \mathsf{ref}) = (t^0 \to t\,\mathsf{ref}, 1, (\{t :: \emptyset\} \cup C, \mathsf{id}))$
  and $\mathsf{Type}(C, \Gamma, !) = ((t\,\mathsf{ref})^0 \to t, 1, (\{t :: \emptyset\} \cup C, \mathsf{id}))$
  and $\mathsf{Type}(C, \Gamma, \mathsf{set}) = ((t\,\mathsf{ref})^0 \to t^0 \to \mathsf{unit}, 1, (\{t :: \emptyset\} \cup C, \mathsf{id}))$
  and $\mathsf{Type}(C, \Gamma, ()) = (\mathsf{unit}, 1, (C, \mathsf{id}))$
  and $\mathsf{Type}(C, \Gamma, \Diamond) = (\Diamond, 1, (C, \mathsf{id}))$.
- If $\mathsf{Type}(C, \Gamma, M) = (\tau_0, \gamma_0, (C_0, \mathsf{S}_0))$
  and $\mathsf{Type}(C_0', \mathsf{S}_0(\Gamma), N) = (\tau_1, \gamma_1, (C_1, \mathsf{S}_1))$ where $C_0' = C_0 \restriction \mathsf{S}_0(\Gamma)$
  and $\{\tau_0 :: \{\ell\}\} \cup C_1 \rhd^* C'$
  then $\mathsf{Type}(C, \Gamma, \langle M, \ell = N\rangle) = (\langle \tau_0, \ell : \tau_1\rangle, \mathsf{S}_1(\gamma_0) \wedge \gamma_1, (C', \mathsf{S}_1\mathsf{S}_0))$.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

- If $\mathsf{Type}(C,\Gamma,M) = (\tau,\gamma,(C_0,\mathsf{S}_0))$
  and $(C',\mathsf{S}) = \mathscr{S}ol(\exists r.\ \{r :: \{\ell\}, t :: \emptyset\} \cup C_0\ ;\ \{\tau = \langle r, \ell : t\rangle\})$
  then $\mathsf{Type}(C,\Gamma,(M.\ell)) = (\mathsf{S}(t),\gamma,(C',\mathsf{S}\mathsf{S}_0))$.
- If $\mathsf{Type}(C,\Gamma,M) = (\tau,\gamma,(C_0,\mathsf{S}_0))$
  and $(C',\mathsf{S}) = \mathscr{S}ol(\exists t.\ \{r :: \{\ell\}, t :: \emptyset\} \cup C_0\ ;\ \{\tau = \langle r, \ell : t\rangle\})$
  then $\mathsf{Type}(C,\Gamma,(M\backslash\ell)) = (\mathsf{S}(r),\gamma,(C',\mathsf{S}\mathsf{S}_0))$.

One can check from this definition that the only serious degree equations that we may have to solve have the form $\alpha = 1$, and arise from the case of a $\mathsf{let\,rec}$ (in the case of application, we may introduce a trivial equation $p = \alpha$). To establish the correctness of the algorithm, we first prove a preliminary lemma.

Lemma 5.9
If $C \vdash \Gamma$ and $\mathsf{Type}(C,\Gamma,M) = (\tau,\gamma,(C',\mathsf{S}))$ then $\mathsf{S} \in \mathscr{S}ub(C \restriction \Gamma, C')$ and $C' \vdash \tau :: \emptyset$.

Proof
By induction on M. Let us examine some cases:

- If $M = x$ then $\Gamma(x) = (\forall C_0.\tau)$ with $\mathsf{dom}(C_0) \cap \mathsf{dom}(C) = \emptyset$, $C' = C \cup C_0$ and $\mathsf{S} = \mathsf{id}$. It is easy to see that $C \vdash (\forall C_0.\tau) :: \emptyset$ implies $C_0, C \vdash \tau :: \emptyset$. Since $C \subseteq C'$, the fact that $\mathsf{id} \in \mathscr{S}ub(C \restriction \Gamma, C')$ is obvious.
- If $M = \lambda x N$, then $C' = \{p \leqslant \delta(x)\} \cup C_0$, $\mathsf{S} = \mathsf{S}_0\backslash t$ and $\tau = \mathsf{S}_0(t)^p \to \theta$ where $\mathsf{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Gamma, N) = (\theta,\delta,(C_0,\mathsf{S}_0))$. By induction hypothesis $\mathsf{S}_0 \in \mathscr{S}ub(\{t :: \emptyset\} \cup C \restriction \Gamma, C_0)$, hence obviously $\mathsf{S} \in \mathscr{S}ub(C \restriction \Gamma, C')$ and $C_0 \vdash \mathsf{S}_0(t) :: \emptyset$. Since $C_0 \vdash \theta :: \emptyset$ by induction hypothesis, we have $C_0 \vdash \tau :: \emptyset$, hence also $C' \vdash \tau :: \emptyset$.
- If $M = (M'N)$ then $\mathsf{S} = \mathsf{S}'\mathsf{S}_1\mathsf{S}_0$ and $\tau = \mathsf{S}'(t)$ with

$$\begin{aligned} \mathsf{Type}(C,\Gamma,M') &= (\tau_0,\gamma_0,(C_0,\mathsf{S}_0)) \\ \mathsf{Type}(C_0',\mathsf{S}_0(\Gamma),N) &= (\tau_1,\gamma_1,(C_1,\mathsf{S}_1)) \\ (C',\mathsf{S}') &= \mathscr{S}ol(\{t :: \emptyset\} \cup C_1\ ;\ \{\mathsf{S}_1(\tau_0) = \tau_1^p \to t\}) \end{aligned}$$

  where $C_0' = C_0 \restriction \mathsf{S}_0(\Gamma)$. By induction hypothesis $\mathsf{S}_0 \in \mathscr{S}ub(C \restriction \Gamma, C_0)$, hence also $\mathsf{S}_0 \in \mathscr{S}ub(C \restriction \Gamma, C_0')$ and $C_0' \vdash \mathsf{S}_0(\Gamma)$. Then by induction hypothesis $\mathsf{S}_1 \in \mathscr{S}ub(C_0', C_1)$, therefore (by Lemma 3.1) $\mathsf{S}_1\mathsf{S}_0 \in \mathscr{S}ub(C \restriction \Gamma, C_1)$. We clearly have

$$\mathsf{S}' \in \mathscr{S}ub(\{t :: \emptyset\} \cup C_1, C')$$

  hence also $\mathsf{S}'\backslash t \in \mathscr{S}ub(C_1, C')$, whence $\mathsf{S} \in \mathscr{S}ub(C \restriction \Gamma, C')$ (for t is fresh, and therefore $t \notin \mathsf{fv}(\mathsf{S}_1\mathsf{S}_0(C))$) and $C' \vdash \mathsf{S}'(t) :: \emptyset$, that is $C' \vdash \tau :: \emptyset$.
- If $M = (\mathsf{let\ rec}\ x = N\ \mathsf{in}\ M')$ then $\mathsf{S} = \mathsf{S}'\mathsf{S}_1\mathsf{S}_0$ with

$$\begin{aligned} \mathsf{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Gamma, N) &= (\tau_0,\gamma_0,(C_0,\mathsf{S}_0)) \\ (C_1,\mathsf{S}_1) &= \mathscr{S}ol(C_0\ ;\ \{\mathsf{S}_0(t) = \tau_0, \gamma_0(x) = 1\}) \\ \mathsf{Type}(C_1 - C_2, \Delta, M') &= (\tau,\gamma_1,(C',\mathsf{S}')) \end{aligned}$$

  where $\Delta = \{x : (\forall C_2.\mathsf{S}_1\mathsf{S}_0(t))\} \cup \mathsf{S}_1\mathsf{S}_0(\Gamma)$ and C_2 is $C_0\backslash\mathsf{fv}(\mathsf{S}_1\mathsf{S}_0(\Gamma))$ if N is pure, and $C_2 = \emptyset$ otherwise. By induction hypothesis, $\mathsf{S}_0 \in \mathscr{S}ub(\{t :: \emptyset\} \cup$

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

$(C \restriction \Gamma), C_0)$, and $\mathsf{S}_1 \in \mathscr{S}ub(C_0, C_1)$, therefore $\mathsf{S}_1\mathsf{S}_0 \in \mathscr{S}ub(\{t :: \emptyset\} \cup (C \restriction \Gamma), C_1)$, from which it follows $C_1 \vdash \{x : \mathsf{S}_1\mathsf{S}_0(t)\} \cup \mathsf{S}_1\mathsf{S}_0(\Gamma)$, and therefore $C_1 - C_2 \vdash \Delta$. By induction hypothesis $C' \vdash \tau :: \emptyset$, and $\mathsf{S}' \in \mathscr{S}ub((C_1 - C_2) \restriction \Delta, C')$. We have $C_1 \vdash \mathsf{S}_1\mathsf{S}_0(t) :: \emptyset$ (see the clause (ii) of Definition 5.1), hence $\mathsf{fv}(\mathsf{S}_1\mathsf{S}_0(t)) \subseteq \mathsf{dom}(C_1)$. By definition of C_2, we then have $(C_1 - C_2) \restriction \Delta = C_1 \restriction \mathsf{S}_1\mathsf{S}_0(\Gamma)$, and therefore $\mathsf{S} \in \mathscr{S}ub(C \restriction \Gamma, C')$ since obviously $\mathsf{S}_1\mathsf{S}_0 \in \mathscr{S}ub(C \restriction \Gamma, C_1 \restriction \mathsf{S}_1\mathsf{S}_0(\Gamma))$.

□

One may observe from this proof that if we start the algorithm with (C, Γ, M) such that $C \vdash \Gamma$, then all the recursive calls to Type operate on arguments (C', Γ', M') such that $C' \vdash \Gamma'$.

Proposition 5.10 (Soundness)
If $\mathsf{Type}(C, \Gamma, M) = (\tau, \gamma, (C', \mathsf{S}))$ with $C \vdash \Gamma$ then $C' ; \mathsf{S}(\Gamma)^\gamma \vdash M : \tau$.

Proof
By induction on M. We examine only some cases.

- If $M = x$, we have $\Gamma(x) = (\forall C_0.\tau)$ with $\mathsf{dom}(C_0) \cap \mathsf{dom}(C) = \emptyset$, $\gamma = 0_x$, $C' = C \cup C_0$, and $\mathsf{S} = \mathsf{id}$. By the previous lemma we have $C' \vdash (\forall C_0.\tau) \geq \tau$, therefore $C' ; \mathsf{S}(\Gamma)^\gamma \vdash x : \tau$.
- If $M = \lambda x N$ then $\tau = \mathsf{S}'(t)^p \to \theta$, $\gamma = 1$ and $C' = \{p \leqslant \delta(x)\} \cup C_0$ where $\mathsf{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Gamma, N) = (\theta, \delta, (C_0, \mathsf{S}'))$ and $\mathsf{S} = \mathsf{S}' \backslash t$. Since t is fresh, and in particular $t \notin \mathsf{fv}(\Gamma)$, we have $\mathsf{S}(\Gamma) = \mathsf{S}'(\Gamma)$. By induction hypothesis $C_0 ; x : \mathsf{S}'(t)^{\delta(x)}, \mathsf{S}(\Gamma)^\delta \vdash N : \theta$, hence $p \leqslant \delta(x), C_0 ; x : \mathsf{S}'(t)^{\delta(x)}, \mathsf{S}(\Gamma)^\delta \vdash N : \theta$ by Lemma 4.2(iii), whence $C' ; \mathsf{S}(\Gamma)^\delta \vdash M : \tau$ by the degree weakening rule, and the typing rule for abstraction.
- If $M = (M'N)$ then $\tau = \mathsf{S}'(t)$, $\gamma = 0_{M'} \wedge \mathsf{S}'(\delta)$ and $\mathsf{S} = \mathsf{S}'\mathsf{S}_1\mathsf{S}_0$ with

$$\begin{aligned} \mathsf{Type}(C, \Gamma, M') &= (\tau_0, \gamma_0, (C_0, \mathsf{S}_0)) \\ \mathsf{Type}(C_0', \mathsf{S}_0(\Gamma), N) &= (\tau_1, \gamma_1, (C_1, \mathsf{S}_1)) \\ (C', \mathsf{S}') &= \mathscr{S}ol(\{t :: \emptyset\} \cup C_1 ; \{\mathsf{S}_1(\tau_0) = \tau_1^p \to t\}) \end{aligned}$$

  where $C_0' = C_0 \restriction \mathsf{S}_0(\Gamma)$, and

$$\delta(x) = \begin{cases} p & \text{if } N = x \\ p \wedge \gamma_1(x) & \text{otherwise} \end{cases}$$

  By induction hypothesis, $C_0' ; \mathsf{S}_0(\Gamma)^{\gamma_0} \vdash M' : \tau_0$, and by Lemma 5.9, $C_0' \vdash \mathsf{S}_0(\Gamma)$. Then by induction hypothesis $C_1 ; \mathsf{S}_1\mathsf{S}_0(\Gamma)^{\gamma_1} \vdash N : \tau_1$. We have

$$C_1 ; \mathsf{S}_1\mathsf{S}_0(\Gamma)^{\mathsf{S}_1(\gamma_0)} \vdash M' : \mathsf{S}_1(\tau_0)$$

  by Lemmas 5.9 and 4.3, and since $\mathsf{S}' \in \mathscr{S}ub(C_1, C')$ we have, by Lemma 4.3 again,

$$C' ; \mathsf{S}(\Gamma)^{\mathsf{S}'\mathsf{S}_1(\gamma_0)} \vdash M' : \mathsf{S}'\mathsf{S}_1(\tau_0)$$

  and

$$C' ; \mathsf{S}(\Gamma)^{\mathsf{S}'(\gamma_1)} \vdash N : \mathsf{S}'(\tau_1)$$

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

Since $\mathsf{S}'\mathsf{S}_1(\tau_0) = \mathsf{S}'(\tau_1)^{\mathsf{S}'(p)} \to \tau$, we conclude using the degree weakening rule, and the typing rule for application.

- If $M = (\mathsf{let\ rec}\ x = N\ \mathsf{in}\ M')$ then $\mathsf{S} = \mathsf{S}'\mathsf{S}_1\mathsf{S}_0$ and

$$\gamma = \begin{cases} \gamma_1(x)_N \wedge \mathsf{S}'\mathsf{S}_1(\gamma_0) \wedge \gamma_1 & \text{if } M' \text{ is not pure} \\ \mathsf{S}'\mathsf{S}_1(\gamma_0) \wedge \gamma_1 & \text{otherwise} \end{cases}$$

with

$$\mathsf{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Gamma, N) \ = \ (\tau_0, \gamma_0, (C_0, \mathsf{S}_0))$$
$$(C_1, \mathsf{S}_1) \ = \ \mathscr{S}ol(C_0 \,;\, \{\mathsf{S}_0(t) = \tau_0, \gamma_0(x) = 1\})$$
$$\mathsf{Type}(C_1 - C_2, \Delta, M') \ = \ (\tau, \gamma_1, (C', \mathsf{S}'))$$

where $\Delta = \{x : (\forall C_2.\mathsf{S}_1\mathsf{S}_0(t))\} \cup \mathsf{S}_1\mathsf{S}_0(\Gamma)$ and C_2 is $C_1 \backslash \mathsf{fv}(\mathsf{S}_1\mathsf{S}_0(\Gamma))$, if N is pure, and $C_2 = \emptyset$ otherwise. Then $C_0 \,;\, (x : \mathsf{S}_0(t), \mathsf{S}_0(\Gamma))^{\gamma_0} \vdash N : \tau_0$, by induction hypothesis, and since $\mathsf{S}_1 \in \mathscr{S}ub(C_0, C_1)$ with $\mathsf{S}_1\mathsf{S}_0(t) = \mathsf{S}_1(\tau_0) = \theta$ and $\mathsf{S}_1(\gamma_0(x)) = 1$, we have $C_1 \,;\, x : \theta^1, \mathsf{S}_1\mathsf{S}_0(\Gamma)^{\mathsf{S}_1(\gamma_0)} \vdash N : \theta$. By Lemma 5.9 we have $C_1 - C_2 \vdash \Delta$, and therefore $\mathsf{S}' \in \mathscr{S}ub((C_1 - C_2) \restriction \Delta, C')$ by Lemma 5.9 again. In particular, S' is the identity on $\mathsf{dom}(C_2)$, and therefore we have

$$C' \,;\, x : (\forall C_2.\mathsf{S}'(\theta))^{\gamma_1(x)}, \mathsf{S}(\Gamma)^{\gamma_1} \vdash M' : \tau$$

by induction hypothesis. Since obviously $\mathsf{S}' \in \mathscr{S}ub(C_1 \restriction \Delta', C' \cup C_2)$ where $\Delta' = \{x : \theta\} \cup \mathsf{S}_1\mathsf{S}_0(\Gamma)$, we have

$$C_2, C' \,;\, x : \mathsf{S}'(\theta)^1, \mathsf{S}(\Gamma)^{\mathsf{S}'(\mathsf{S}_1(\gamma_0))} \vdash N : \mathsf{S}'(\theta)$$

by Lemmas 4.4 and 4.3 (and possibly Lemma 4.2 if $\mathsf{dom}(C') \cap \mathsf{dom}(C_2) \neq \emptyset$). We conclude using the degree weakening rule, and the typing rule for the $\mathsf{let\ rec}$ construct.

□

Proposition 5.11 (Completeness)
Let Γ and M be such that $C' \,;\, \mathsf{S}(\Gamma)^\delta \vdash M : \tau$ for some type τ, degree assignment δ and substitution $\mathsf{S} \in \mathscr{S}ub(C, C')$ with $C \vdash \Gamma$ and $C \restriction \Gamma = C$. Then $\mathsf{Type}(C, \Gamma, M) = (\theta, \gamma, (C_0, \mathsf{S}_0))$, and there exists a substitution $\mathsf{S}' \in \mathscr{S}ub(C_0, C')$ such that $\mathsf{S} = \mathsf{S}'\mathsf{S}_0$, $\tau = \mathsf{S}'(\theta)$ and $C' \vdash \delta \leqslant \mathsf{S}'(\gamma)$.

Proof
By induction on the inference of $C' \,;\, \mathsf{S}(\Gamma)^\delta \vdash M : \tau$, and by case on the last rule used to infer this sequent. The case where this rule is degree weakening is trivial, and is omitted. Then the proof actually proceeds by induction on M. We only examine some cases:

- If $M = x$ then $\mathsf{S}(\Gamma)^\delta = x : \sigma^0, \Gamma$ with $C' \vdash \sigma \geq \tau$, that is $\sigma = (\forall C_1.\tau_1)$ with $\tau = \mathsf{S}_1(\tau_1)$ for $\mathsf{S}_1 \in \mathscr{S}ub(C_1, C')$. We may assume that $\mathsf{dom}(C_1) \cap (\mathsf{dom}(C) \cup \mathsf{dom}(C')) = \emptyset$, and therefore $\Gamma(x) = (\forall C_1.\tau_0)$ with $\tau_1 = \mathsf{S}(\tau_0)$. Then $\mathsf{Type}(C, \Gamma, M) = (\tau_0, 0_x, (C \cup C_1, \mathsf{id}))$. We may let $\mathsf{S}' = \mathsf{S}_1 \cup \mathsf{S}$ in this case.
- If $M = \lambda x N$ then $\tau = \tau_0^d \to \tau_1$ and $\delta = 1$ with $C' \,;\, x : \tau_0^d, \mathsf{S}(\Gamma)^{\delta'} \vdash N : \tau_1$ for some δ'. If we let $\Delta = \{x : t\} \cup \Gamma$ where t is fresh, and $\mathsf{S}_1 = \{t \mapsto \tau_0\} \cup \mathsf{S}$, then by

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

induction hypothesis $\mathsf{Type}(\{t :: \emptyset\} \cup C, \Delta, N) = (\theta, \gamma, (C_0, \mathsf{S}_0))$ and there exists $\mathsf{S}'' \in \mathscr{S}ub(C_0, C')$ such that $\mathsf{S}_1 = \mathsf{S}''\mathsf{S}_0$, $\tau_1 = \mathsf{S}''(\theta)$ and $C' \vdash \{x \mapsto d\} \cup \delta' \leqslant \mathsf{S}''(\gamma)$. In particular $\tau_0 = \mathsf{S}''(\mathsf{S}_0(t))$ and $C' \vdash d \leqslant \mathsf{S}''(\gamma(x))$. We have

$$\mathsf{Type}(C, \Gamma, M) = (\mathsf{S}_0(t)^p \to \theta, 1, (\{p \leqslant \gamma(x)\} \cup C_0, \mathsf{S}_0 \backslash t))$$

We let $\mathsf{S}' = \{p \mapsto d\} \cup \mathsf{S}''$ in this case.

- If $M = (M'N)$ then $C' ; \mathsf{S}(\Gamma)^{\delta'} \vdash M' : \theta^a \to \tau$ and $C' ; \mathsf{S}(\Gamma)^{\delta'} \vdash N : \theta$ with $\delta = 0_{M'} \wedge \delta''$ where

$$\delta''(x) = \begin{cases} a & \text{if } N = x \\ a \wedge \delta'(x) & \text{otherwise} \end{cases}$$

Then by induction hypothesis $\mathsf{Type}(C, \Gamma, M') = (\tau_0, \gamma_0, (C_0, \mathsf{S}_0))$ and there exists $\mathsf{S}'_0 \in \mathscr{S}ub(C_0, C')$ such that $\mathsf{S} = \mathsf{S}'_0\mathsf{S}_0$, $\theta^a \to \tau = \mathsf{S}'_0(\tau_0)$ and $C' \vdash \delta' \leqslant \mathsf{S}'_0(\gamma_0)$. By Lemma 5.9 $C_0 \vdash \mathsf{S}_0(C)$, and therefore by induction hypothesis $\mathsf{Type}(C'_0, \mathsf{S}_0(\Gamma), N) = (\tau_1, \gamma_1, (C_1, \mathsf{S}_1))$ where $C'_0 = C_0 \restriction \mathsf{S}_0(\Gamma)$, and there exists $\mathsf{S}'_1 \in \mathscr{S}ub(C_1, C')$ such that $\mathsf{S}'_0 = \mathsf{S}'_1\mathsf{S}_1$, $\theta = \mathsf{S}'_1(\tau_1)$ and $C' \vdash \delta' \leqslant \mathsf{S}'_1(\gamma_1)$. Then the verification condition $(\{t :: \emptyset\} \cup C_1 ; \{\mathsf{S}_1(\tau_0) = \tau_1^p \to t\})$ has a solution. Let (C'', S'') be its most general solution, so that $\{t \mapsto \tau\} \cup \mathsf{S}'_1 = \mathsf{S}''_1\mathsf{S}''$. Then

$$\mathsf{Type}(C, \Gamma, (M'N)) = (\mathsf{S}''(t), 0_{M'} \wedge \mathsf{S}''(\gamma), (C'', \mathsf{S}''\mathsf{S}_1\mathsf{S}_0))$$

where

$$\gamma(x) = \begin{cases} p & \text{if } N = x \\ p \wedge \gamma_1(x) & \text{otherwise} \end{cases}$$

We have $\mathsf{S} = \mathsf{S}''_1\mathsf{S}''\mathsf{S}_1\mathsf{S}_0$, $\mathsf{S}''_1\mathsf{S}''(t) = \tau$ and $\mathsf{S}''_1\mathsf{S}''(p) = a$, and it is easy to check that $C' \vdash \delta \leqslant \mathsf{S}''_1(0_{M'} \wedge \mathsf{S}''(\gamma))$.

- If $M = (\mathsf{let\ rec}\ x = N\ \mathsf{in}\ M')$ then we have $C'', C' ; x : \theta^1, \mathsf{S}(\Gamma)^{\delta'} \vdash N : \theta$ and $C' ; x : (\forall C''.\theta)^\alpha, \mathsf{S}(\Gamma)^{\delta'} \vdash M' : \tau$, where $t \in \mathsf{dom}(C'') \Rightarrow t \notin \mathsf{dom}(C')$ and C'' is empty if N is not pure, and

$$\delta = \begin{cases} \alpha_N \wedge \delta' & \text{if } M \text{ is not pure} \\ \delta' & \text{otherwise} \end{cases}$$

By induction hypothesis

$$\mathsf{Type}(\{t :: \emptyset\} \cup C, \{x : t\} \cup \Gamma, N) = (\tau_0, \gamma_0, (C_0, \mathsf{S}_0))$$

and there exists $\mathsf{S}'_0 \in \mathscr{S}ub(C_0, C')$ such that $\{t \mapsto \theta\} \cup \mathsf{S} = \mathsf{S}'_0\mathsf{S}_0$, $\theta = \mathsf{S}'_0(\tau_0)$, $C' \vdash 1 \leqslant \mathsf{S}'_0(\gamma_0(x))$ and $C' \vdash \delta' \leqslant \mathsf{S}'_0(\gamma_0)$. Then $(C_0 ; \{\mathsf{S}_0(t) = \tau_0, \gamma_0(x) = 1\})$ has a solution. Let (C_1, S_1) be its most general solution, so that $\mathsf{S}'_0 = \mathsf{S}'_1\mathsf{S}_1$, and let C_2 be $C_1 \backslash \mathsf{fv}(\mathsf{S}_1\mathsf{S}_0(\Gamma))$, if N is pure, and $C_2 = \emptyset$ otherwise. We have $C' \vdash (\forall C_2.\mathsf{S}_1\mathsf{S}_0(t)) \geq (\forall C''.\theta)$, hence $C' ; x : (\forall C_2.\mathsf{S}_1\mathsf{S}_0(t))^\alpha, \mathsf{S}(\Gamma)^{\delta'} \vdash M' : \tau$ by Lemma 4.8. Then by induction hypothesis

$$\mathsf{Type}(C_1 - C_2, \{x : (\forall C_2.\mathsf{S}_1\mathsf{S}_0(t))\} \cup \mathsf{S}_1\mathsf{S}_0(\Gamma), M) = (\tau_1, \gamma_1, (C', \mathsf{S}'))$$

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

and there exists S'' such that $\mathsf{S}'_1 = \mathsf{S}''\mathsf{S}'$, $\tau = \mathsf{S}''(\tau_1)$, $C' \vdash \alpha \leqslant \mathsf{S}''(\gamma_1(x))$ and $C' \vdash \delta' \leqslant \mathsf{S}''(\gamma_1)$. Finally we have $\mathsf{Type}(C, \Gamma, M) = (\tau_1, \gamma, (C', \mathsf{S}'\mathsf{S}_1\mathsf{S}_0))$ where

$$\gamma = \begin{cases} \gamma_1(x)_N \wedge \mathsf{S}'\mathsf{S}_1(\gamma_0) \wedge \gamma_1 & \text{if } M \text{ is not pure} \\ \mathsf{S}'\mathsf{S}_1(\gamma_0) \wedge \gamma_1 & \text{otherwise} \end{cases}$$

It is easy to check that $C' \vdash \delta \leqslant \mathsf{S}''(\gamma)$.

□

Finally, combining the soundness and completeness properties, we get our second main result.

Theorem 5.12 (Type Assignment of a Principal Type)
If there is no C such that $C \vdash \Gamma$, or if $C \vdash \Gamma$ and $\mathsf{Type}(C, \Gamma, M)$ fails, then for any degree assignment γ, the expression M is not typable in the context $C\,;\Gamma^{\gamma}$. If Γ is closed, that is $\mathsf{fv}(\Gamma) = \emptyset$, and $C\,;\Gamma^{\delta} \vdash M : \tau$ for some C, δ and τ, then $\mathsf{Type}(\emptyset, \Gamma, M)$ succeeds, returning $(\theta, \gamma, (C_0, \mathsf{id}))$, so that $C_0\,;\Gamma^{\gamma} \vdash M : \theta$, and there exists $\mathsf{S} \in \mathscr{S}ub(C_0, C)$ such that $\tau = \mathsf{S}(\theta)$ and $C \vdash \delta \leqslant \mathsf{S}(\gamma)$.

One may observe that, thanks to Lemma 5.5, there is a C such that $C \vdash \Gamma$ if and only if there exists C such that $\mathscr{C}(\Gamma) \rhd^* C$ where $\mathscr{C}(\Gamma)$ is the set of annotation assertions to validate for Γ to be acceptable, that is $\mathscr{C}(x_1 : \sigma_1, \ldots, x_n : \sigma_n) = \{\sigma_1 :: \emptyset, \ldots, \sigma_n :: \emptyset\}$.

6 Object-oriented programming

In this section, we illustrate the expressive power of our calculus, as regards object-orientation, both from an operational and from a typing point of view. To this end, we will introduce a few derived constructs. Our approach to object-orientation is *mixin-based*. The notion of a "mixin" has been introduced in object-oriented programming languages of the 1980s, mainly in languages based on Lisp. In Bracha & Cook (1990), it has been advocated as the "building block" for inheritance, and has recently received some attention (see, for instance, Ancona & Zucca (1998), Bono *et al.* (1999b) and Flatt *et al.* (1998)). Roughly speaking, a mixin is a class definition parameterized over its superclass (there is some similarity with the parameterized classes of Eiffel (Meyer, 1986) and the "virtual classes" of Beta (Madsen & Møller Pedersen, 1989)). Let us introduce some informal terminology:

- An *object* is the fixpoint ($\mathsf{fix}\, Gen$) of a generator (Cook & Palsberg, 1989).
- A *generator* is a function of a "self" parameter, returning a record of fields and methods. Then a typical generator value is thus

$$\lambda s\langle \cdots fields \cdots methods \cdots \rangle$$

like the empty generator $\lambda s\langle\rangle$, and an object is therefore a recursive record, as in Cardelli (1984), Cook *et al.* (1994), Snyder (1986a) and Wand (1994). Notice that for a generator to be able to generate some object, the type of self (that is, s) should be unifiable with the type of the record returned by the generator, and moreover the generator must be a protective function.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

- A field is like any ordinary field in a record, either mutable or not. In an object generator, a field is not supposed to contain the self parameter.
- A *method* is a field in a record, whose value is a "thunk", that is a function of a dummy parameter (just to freeze evaluation at this point). A method explicitly depends upon two parameters, super and self, representing respectively the current object as a member of the superclass, and as a member of the current class, as far as the ownership of methods is concerned (these parameters are not keywords; rather, they are bound names, subject to α-conversion). We use a special syntax for invoking a method ℓ of an object, different from selection, namely $(M \Leftarrow \ell)$ (just to unfreeze evaluation).
- A *mixin* is a function mapping generators to generators, usually by extending and/or modifying the record returned by its argument. A typical mixin value is thus $\lambda g\, \lambda s \langle gs, \ldots \rangle$, and inheritance is basically mixin composition, that is $M = \lambda g\, M''(M'\, g)$ if M inherits M', with modification (which is a mixin) M''. In the methods introduced by a mixin, the super parameter is to be interpreted as $(g\, s)$, the "generic object" (where self is not yet bound) of the superclass, while the self parameter is obviously interpreted as s.
- A *class* is a function taking as argument a series of "instance parameters" and returning a mixin([3]). A typical class value is thus

$$\lambda x_1 \ldots x_n . \lambda g\, \lambda s \langle gs, \ldots \mathit{fields} \ldots \mathit{methods} \ldots \rangle$$

  An object instance of such a class C is the fixpoint of the generator obtained by applying the class to initial values of the instance parameters, usually determining the initial state of the object, and to the empty generator, that is:

$$\mathsf{new}(C\, N_1 \cdots N_n) = \mathsf{fix}(C\, N_1 \cdots N_n\, (\lambda s \langle\rangle))$$

  If the class, or more appropriately the parameterized mixin C has no instance, because the type of self is not unifiable with the type of the record returned by the generator, we say that the class is *abstract*. To inherit a class, one has to "extract" from it the mixin it returns. This is usually done by applying the class to formal parameters, i.e. $(C\, y_1 \cdots y_n)$, which are instance parameters of the subclass, but one may more generally inherit the class as $(C\, N_1 \cdots N_n)$.

We must again point out the fact that such an approach to object-oriented programming constructs would not be possible without having extended recursive definitions to allow defining non-functional recursive values. Indeed, we have

$$\mathsf{new}(C\, N_1 \cdots N_n) \overset{*}{\rightarrow} (\mathsf{let\ rec}\ x = (\lambda s \langle \cdots \rangle) x\ \mathsf{in}\ x)$$

and we do not know of any call-by-value language where such a let rec is allowed. Nevertheless, and although this section is devoted to demonstrate, by means of examples, the increase in expressiveness we gain by considering such a non-standard

[3] This notion of a class is slightly non-standard, since there are some mixins that one would not normally call "classes". Indeed, the name "mixin" is sometimes used only for classes that are not intended to be instantiated into objects.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

$$
\begin{array}{rcll}
M, N \ldots & ::= & \cdots \mid (M \Leftarrow \ell) & \textit{method invocation} \\
V, W \ldots & ::= & \cdots \mid \mathsf{new} \mid \mathsf{mixin}(T) & \\
T & ::= & & \textit{mixins} \\
 & \mid & \mathsf{var}\ \ell = N \mid \mathsf{cst}\ \ell = N & \textit{field definition} \\
 & \mid & \mathsf{meth}\ \ell(y,x) = M \mid \mathsf{meth}\ \ell(y,x) \leftarrow M & \textit{method definition/override} \\
 & \mid & \mathsf{without}\ \ell \mid \mathsf{rename}\ \ell\ \mathsf{as}\ \ell' & \textit{restriction, renaming} \\
 & \mid & \mathsf{inherit}\ M \mid (T, T') & \textit{inheritance}
\end{array}
$$

Fig. 10. Mixin constructs.

$$
\begin{array}{rcl}
[\![M \Leftarrow \ell]\!] & = & ([\![M]\!].\ell)() \\
[\![\mathsf{new}]\!] & = & \lambda m(\mathsf{fix}(m\, \lambda s \Diamond)) \\
[\![\mathsf{mixin}(T)]\!] & = & [\![T]\!] \\
[\![\mathsf{var}\ \ell = N]\!] & = & \lambda g \lambda s \langle gs, \ell = \mathsf{ref}\ [\![N]\!] \rangle \\
[\![\mathsf{cst}\ \ell = N]\!] & = & \lambda g \lambda s \langle gs, \ell = [\![N]\!] \rangle \\
[\![\mathsf{meth}\ \ell(y,x) = M]\!] & = & \lambda g \lambda s (\mathsf{let}\ z = gs\ \mathsf{in}\ \langle z, \ell = \lambda.((\lambda y \lambda x [\![M]\!]) z\, s) \rangle) \\
[\![\mathsf{meth}\ \ell(y,x) \leftarrow M]\!] & = & \lambda g \lambda s (\mathsf{let}\ z = gs\ \mathsf{in}\ \langle z \backslash \ell, \ell = \lambda.((\lambda y \lambda x [\![M]\!]) z\, s) \rangle) \\
[\![\mathsf{without}\ \ell]\!] & = & \lambda g \lambda s ((gs) \backslash \ell) \\
[\![\mathsf{rename}\ \ell\ \mathsf{as}\ \ell']\!] & = & \lambda g \lambda s (\mathsf{let}\ z = gs\ \mathsf{in}\ \langle z \backslash \ell, \ell' = z.\ell \rangle) \\
[\![\mathsf{inherit}\ M]\!] & = & \lambda g \lambda s (([\![M]\!] \lambda s (g\, s)) s) \\
[\![T, T']\!] & = & \lambda g \lambda s (([\![T']\!] \lambda s (([\![T]\!] \lambda s (gs)) s)) s)
\end{array}
$$

Fig. 11. Interpretation.

fixpoint, we shall not pay very much attention to degrees here: they just let everything go smoothly.

Having thus informally described our model for objects and inheritance, we now introduce some corresponding syntax to define mixins and objects, extending the language as indicated in figure 10. We could also add a notation for field override, but this can actually be written ($\mathsf{without}\ \ell, \mathsf{var}\ \ell = N'$) or ($\mathsf{without}\ \ell, \mathsf{cst}\ \ell = N'$). We use the same notation for method override than for overriding a field in a record, although these are not exactly the same operations. The interpretation of the extended language into the core language is given in figure 11, by means of a translation $[\![\cdot]\!]$ which is an isomorphism as regards the core constructs. In this figure, we use $\lambda.M$ to denote a thunk, that is an abstraction over a variable which does not occur in M. In the translation of mixins, the variables g, s and z are supposed to be fresh. Then one can see that, in particular, a field has no knowledge of the self parameter, since s does not occur in $[\![N]\!]$, whereas the super and self parameters of a method are instantiated into s and z, with $z = gs$, respectively. We have adopted a sophisticated interpretation of $\mathsf{inherit}\ M$ and (T, T'), involving η-expansion, to avoid using these constructs with arguments of inappropriate type. However, one should understand $\mathsf{inherit}\ M$ simply as $\lambda g([\![M]\!]g)$ (or even $[\![M]\!]$, but notice that a mixin has to be a value), and similarly (T, T') is to be understood as $\lambda g([\![T']\!]([\![T]\!]g))$, that is $[\![T']\!] \circ [\![T]\!]$. Then η-expansion is only used to ensure that

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

these constructs have types of the form $(\theta_0^a \to \tau_0)^b \to \theta_1^c \to \tau_1$, and not just $\theta^d \to \tau$. We shall come back to this point below.

Regarding the derived typing of the extended language, we first observe that, since it is a thunk, a method in a mixin always has a type of the form $(t^p \to \tau)$ (with no constraint on p), and that when invoking this method, the type variable t will be instantiated into unit([4]). Then we may introduce the notation $\langle \rho, \mathsf{meth}\ \ell : \tau \rangle$, standing for $\langle \rho, \ell : \mathsf{unit}^p \to \tau \rangle$, and one can see that a derived typing for method invocation is – assuming that $\gamma(x) = 1$ for $x \notin \mathsf{fv}(M)$:

$$\frac{C\,;\Gamma^\gamma \vdash [\![M]\!] : \langle \rho, \mathsf{meth}\ \ell : \tau \rangle}{C\,;\Gamma^{0_M} \vdash (M \Leftarrow \ell) : \tau}$$

Regarding the combinator new that is intended to instantiate a class, we have

$$\frac{}{C\,;\Gamma^\gamma \vdash \mathsf{new} : ((\theta^p \to \langle\!\rangle)^q \to \tau^1 \to \tau)^0 \to \tau}$$

The derived typing of the mixin constructs is given in figure 12, which we now comment. A first observation is that the type of a mixin is generally quite big. Although this is clearly very important from a pragmatic point of view, we shall not in this paper attempt to introduce meaningful abbreviations regarding these types – except for the type of methods – since our purpose is mainly to experiment with a preliminary language design.

Thanks to the sophisticated interpretation of $\mathsf{inherit}\ M$ and (T, T'), a mixin has a type of the form $(\theta_0^a \to \tau_0)^b \to \theta_1^c \to \tau_1$. Moreover, apart from pathological uses of $\mathsf{inherit}\ M$, we generally have – this will be the case for all the examples below – $\theta_0 = \theta_1$ so that this type, denoted θ, is the type of self, $(\theta^a \to \tau_0)$ is the type of the generator argument (associated with the superclass), τ_0 is the type of the super parameter, and τ_1 is the type of the value returned by the mixin. In the examples that we shall examine, θ and τ_1 are "open" record types, that is $\langle t, \ell_1 : \vartheta_1, \ldots, \ell_n : \vartheta_n \rangle$. In most models of typed objects, the latter usually is a "fixed" record type, that is $\langle \ell'_1 : \vartheta'_1, \ldots, \ell'_n : \vartheta'_n \rangle$, taken as representing the type, or the interface of the class. In our model, a class – being a mixin – depends on a superclass whose value is only fixed at object creation time, and this explains the "open" type. We must also point out that, in most models of typed objects, except that of Wand (1994) that we follow (see also Eifrig *et al.* (1995b)), the type of self is supposed to be a subtype of the type of the class. In particular, self is generally supposed to support all the methods offered by the class. Since we are inferring types of variables from their usage in expressions, this will generally not be the case here, and therefore the type of self is a useful information to know about the type of a mixin. Notice that for a class to have some instance, one must be able to solve the equation $\theta = \tau_1$, as required in the type of new. We could at compile time declare the class to be abstract if this equation has no solution. On the other hand, except for pathological uses of $\mathsf{inherit}\ M$ (e.g. with $M = \lambda x \lambda y\, y$), a mixin is always a protective function of self

[4] In the implementation, the dummy parameter of a method is forced to have type unit, and therefore any method has a type of the form $(\mathsf{unit}^p \to \tau)$.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

$$\frac{C\,;\Gamma^{\gamma}\vdash[\![N]\!]:\tau \qquad C\vdash\rho::\{\ell\} \qquad C\vdash c\leqslant a}{C\,;\Gamma^{1}\vdash \mathsf{var}\;\ell=N:(\theta^{a}\to\rho)^{b}\to\theta^{c}\to\langle\rho,\ell:\tau\;\mathsf{ref}\rangle}$$

$$\frac{C\,;\Gamma^{\gamma}\vdash[\![N]\!]:\tau \qquad C\vdash\rho::\{\ell\} \qquad C\vdash c\leqslant a}{C\,;\Gamma^{1}\vdash \mathsf{cst}\;\ell=N:(\theta^{a}\to\rho)^{b}\to\theta^{c}\to\langle\rho,\ell:\tau\rangle}$$

$$\frac{C\,;y:\rho^{a'},x:\theta^{b'},\Gamma^{\gamma}\vdash[\![M]\!]:\tau \qquad C\vdash\rho::\{\ell\} \qquad C\vdash c\leqslant a}{C\,;\Gamma^{1}\vdash \mathsf{meth}\;\ell(y,x)=M:(\theta^{a}\to\rho)^{b}\to\theta^{c}\to\langle\rho,\mathsf{meth}\;\ell:\tau\rangle}$$

$$\frac{C\,;y:\langle\rho,\ell:\tau'\rangle^{a'},x:\theta^{b'},\Gamma^{\gamma}\vdash[\![M]\!]:\tau \qquad C\vdash\rho::\{\ell\} \qquad C\vdash c\leqslant a}{C\,;\Gamma^{1}\vdash \mathsf{meth}\;\ell(y,x)\leftarrow M:(\theta^{a}\to\langle\rho,\ell:\tau'\rangle)^{b}\to\theta^{c}\to\langle\rho,\mathsf{meth}\;\ell:\tau\rangle}$$

$$\frac{C\vdash\rho::\{\ell\} \qquad C\vdash c\leqslant a}{C\,;\Gamma^{1}\vdash \mathsf{without}\;\ell:(\theta^{a}\to\langle\rho,\ell:\tau\rangle)^{b}\to\theta^{c}\to\rho}$$

$$\frac{C\vdash\rho::\{\ell,\ell'\} \qquad C\vdash c\leqslant a}{C\,;\Gamma^{1}\vdash \mathsf{rename}\;\ell\;\mathsf{as}\;\ell':(\theta^{a}\to\langle\rho,\ell:\tau\rangle)^{b}\to\theta^{c}\to\langle\rho,\ell':\tau\rangle}$$

$$\frac{C\,;\Gamma^{\gamma}\vdash[\![M]\!]:(\theta_0^{a}\to\tau_0)^{b}\to\theta_1^{c}\to\tau_1 \qquad C\vdash a\leqslant a',c'\leqslant c}{C\,;\Gamma^{1}\vdash \mathsf{inherit}\;M:(\theta_0^{a'}\to\tau_0)^{b'}\to\theta_1^{c'}\to\tau_1}$$

$$\frac{\begin{array}{l}C\,;\Gamma^{\gamma}\vdash[\![T]\!]:(\theta_0^{a}\to\tau_0)^{b}\to\theta_1^{c}\to\tau_1\\ C\,;\Gamma^{\gamma}\vdash[\![T']\!]:(\theta_1^{a'}\to\tau_1)^{b'}\to\theta_2^{c'}\to\tau_2\\ C\vdash c''\leqslant c',a'\leqslant c,a\leqslant a''\end{array}}{C\,;\Gamma^{1}\vdash(T,T'):(\theta_0^{a''}\to\tau_0)^{b''}\to\theta_2^{c''}\to\tau_2}$$

Fig. 12. Typing the mixins.

(provided that the super-generator is protective too, which is obviously the case of the "universal" generator $\lambda s\langle\rangle$), since the self parameter is only used in method bodies, which are values. Therefore, if we only use as arguments of inherit functions that are protective with respect to their second argument, we do not end up with an unsafe recursion when trying to create an object instance of a class. To avoid any pathological use of inherit, we could also have considered inherit(M), for each expression M, as a new primitive value, with $(\mathsf{inherit}(M)V)\to(MV)$ and

$$\frac{C\,;\Gamma^{\gamma}\vdash M:(\theta^{a}\to\tau)^{b}\to\theta^{1}\to\tau}{C\,;\Gamma^{\gamma}\vdash \mathsf{inherit}(M):(\theta^{a}\to\tau)^{b}\to\theta^{1}\to\tau}$$

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

As a last comment about the typing of mixins, we can now explain why we made two cases in the typing rule for (let $x = N$ in M), according as to whether M is pure or not: observe that in the translations of meth $\ell(y,x) = M'$, meth $\ell(y,x) \leftarrow M'$ and rename ℓ as ℓ', we have such a (let $x = N$ in M) construct where M is pure. Then, although z has degree 0 in these expressions, we can still assume that the self parameter s is protected – provided that g is protective, whence $C \vdash c \leqslant a$ in the derived rules – and this would not hold without a specific case for M pure in the typing rule for the let construct. For this reason, a first version of our type system, presented in Boudol (2001), was unable to accept objects instances of a class where the meth $\ell(y,x) \leftarrow M$ or rename ℓ as ℓ' constructs were used.

Now let us see examples illustrating the use of these derived constructs. Obviously, we do not claim that the examples we propose are interesting programs by themselves; they are just meant to give an idea of the flexibility of the approach. They are all variations around the standard example in discussing models of objects, namely the "point". We start by rewriting in our syntax the class of points that we gave in Section 2 – using the standard notation for assignment, that is $M := N$ instead of set $M\,N$:

$$\begin{array}{l}\mathsf{let\ point} = \lambda x\, \mathsf{mixin}\,(\\ \qquad\qquad \mathsf{var\ pos} = x,\\ \qquad\qquad \mathsf{meth\ move}(z,s) = \lambda d(s.\mathsf{pos} := \mathsf{!}s.\mathsf{pos} + d)\\ \qquad\qquad\quad)\\ \mathsf{in}\ \cdots\end{array}$$

One can see that, up to some β_{v}-conversions, the translation of the mixin defining the points is as follows:

$$\begin{array}{l}\lambda x \lambda g \lambda s \langle g s, \mathsf{pos} = \mathsf{ref}\ x,\\ \qquad\qquad \mathsf{move} = \lambda.\lambda d(s.\mathsf{pos} := \mathsf{!}s.\mathsf{pos} + d)\rangle\end{array}$$

which is very similar to our definition of the point class in Section 2, except for the g parameter, and the record gs that this mixin extends. Then, again up to some β_{v}-conversions, the expression $(\mathsf{point}\,0)\lambda s\langle\rangle$ evaluates into

$$\begin{array}{l}\lambda s\langle \mathsf{pos} = u,\\ \qquad \mathsf{move} = \lambda.\lambda d(s.\mathsf{pos} := \mathsf{!}s.\mathsf{pos} + d)\rangle\end{array}$$

where u is a location whose value in the store is 0, and therefore the object $\mathsf{fix}((\mathsf{point}\,0)\lambda s\langle\rangle)$, that is $\mathsf{new}(\mathsf{point}\,0)$, is the recursive record

$$(\mathsf{let\ rec}\ s = \langle \mathsf{pos} = u, \mathsf{move} = \lambda.\lambda d(s.\mathsf{pos} := \mathsf{!}s.\mathsf{pos} + d)\rangle\ \mathsf{in}\ s)$$

Before we examine the typing of such an object, we notice that, from an operational point of view, our representation of objects allows some operations that would not

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

be available using a self-application semantics, like

$$(\mathsf{let}\ p = (\mathsf{new}(\mathsf{point}\,0))\backslash \mathsf{pos}\ \mathsf{in}\ (p \Leftarrow \mathsf{move})42)$$

Regarding the typing, one can see that, abbreviating $(\forall t :: \emptyset.\sigma)$ into $(\forall t.\sigma)$, the (polymorphic) type of point is – assuming that $+$ is of type $\mathsf{int}^0 \to \mathsf{int}^0 \to \mathsf{int}$:

$$\begin{aligned} r' \leqslant r; \vdash \mathsf{point}: \\ \forall t_0.\forall t_1 :: \{\mathsf{pos}\}.\forall t_2 :: \{\mathsf{pos}, \mathsf{move}\}.t_0{}^p \to \\ (\rho^r \to t_2)^q \to \\ \rho^{r'} \to \\ \langle t_2, \mathsf{pos}: t_0\,\mathsf{ref}, \mathsf{meth}\ \mathsf{move}: \mathsf{int}^0 \to \mathsf{unit}\rangle \end{aligned}$$

where ρ is the type of self, namely $\rho = \langle t_1, \mathsf{pos}: \mathsf{int}\,\mathsf{ref}\rangle$, and therefore we have:

$$\begin{aligned} \vdash (\mathsf{point}\,0)\lambda s\langle\rangle: \ & \langle t_1, \mathsf{pos}: \mathsf{int}\,\mathsf{ref}\rangle^{r'} \to \\ & \langle \mathsf{pos}: \mathsf{int}\,\mathsf{ref}, \mathsf{meth}\ \mathsf{move}: \mathsf{int}^0 \to \mathsf{unit}\rangle \end{aligned}$$

We finally get the expected type for the point object:

$$\vdash \mathsf{new}(\mathsf{point}\,0): \ \langle \mathsf{pos}: \mathsf{int}\,\mathsf{ref}, \mathsf{meth}\ \mathsf{move}: \mathsf{int}^0 \to \mathsf{unit}\rangle$$

It is worth noting (cf. Wand (1994)) that the type of self only retains what is required of the current object from its usage in the definition of the class, namely the presence of a pos field with an int ref type. In particular, the type $\langle t_1, \mathsf{pos}: \mathsf{int}\,\mathsf{ref}\rangle$ of self is not a subtype of the "interface" of the point class, that we may represent with $\langle \mathsf{pos}: t_0\,\mathsf{ref}, \mathsf{meth}\ \mathsf{move}: \mathsf{int}^0 \to \mathsf{unit}\rangle$ – this is the type of $\lambda x\lambda s(\mathsf{point}\,x(\lambda s\langle\rangle)s)$. This allows us to make use of some non-standard forms of code reuse, like for instance *inheritance by restriction*:

$$\begin{aligned} \mathsf{immobilePoint} = \lambda x\,\mathsf{mixin}\,(\\ \mathsf{inherit}(\mathsf{point}\,x), \\ \mathsf{without}\ \mathsf{move} \\ ) \end{aligned}$$

Here we inherit a point object, with a formal initial position x, and we decide to introduce a new kind of points, which we cannot move. The translation of this class is, again with some optimisation:

$$\lambda x\lambda g\lambda s\langle gs, \mathsf{pos} = \mathsf{ref}\ x\rangle$$

Then we can create an object instance of that class, because the type of self, which is still $\langle t_1, \mathsf{pos}: \mathsf{int}\,\mathsf{ref}\rangle$, may be unified with the type of the record returned by the class. This particular example may not look especially interesting – we shall see later another example of the use of restriction. Some similar examples of excluding methods, like for instance building a class of stacks from a given class of dequeues, were given long ago by Snyder (1986a, 1986b). Clearly, allowing such a reuse mechanism reinforces the fact that "inheritance is not subtyping" (Cook *et al.*, 1994).

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

The fact that the type of self in the point class does not contain any reference to the move method can be exploited in another way: we may redefine this method with a type which is unrelated (except for what it requires of self) to the one it has in the point class. For instance, one may decide to modify the move method, so that it now takes one further argument, which is a unit of measure. Then we write, assuming a function f giving the conversion factor into the default unit:

$$\begin{array}{l}\mathsf{uPoint} = \lambda x\,\mathsf{mixin}\,(\\ \qquad \mathsf{inherit}(\mathsf{point}\,x),\\ \qquad \mathsf{meth}\ \mathsf{move}(z,s) \leftarrow \lambda d \lambda y(s.\mathsf{pos} := !s.\mathsf{pos} + d * (f\ y))\\ \qquad\ )\end{array}$$

The type of self in this class is the same as in point, whereas the (abbreviated) type of move is now $\mathsf{int}^0 \to \vartheta \to \mathsf{unit}$ where ϑ is the type of measure units. This is an example of *inheritance by method override* (or *redefinition*). Usually, in this kind of inheritance, the redefined method is required to have the same type as the overriden method, or a subtype of it. Let us see now an example of the use of the super parameter in methods. We decide to create a new kind of points, similar to the previous one, but using the unit of measure as an instance parameter, fixing the scale for its movements:

$$\begin{array}{l}\mathsf{scaledPoint} = \lambda x \lambda y\,\mathsf{mixin}\,(\\ \qquad \mathsf{inherit}(\mathsf{uPoint}\,(x * (f\ y))),\\ \qquad \mathsf{meth}\ \mathsf{move}(z,s) \leftarrow \lambda d(z \Leftarrow \mathsf{move}\,d\ y)\\ \qquad\ )\end{array}$$

Here we inherit the uPoint class with an argument which is not just an instance parameter, as in the previous examples, but a compound value. Notice that while we used, in methods, the record selection syntax $M.\ell$ when accessing the field of an object, we must obviously use the derived construct $M \Leftarrow \ell$ to invoke a method of an object – namely, in this example, invoking the previous version of the move method, attached to the superclass. The next example uses the very common form of *inheritance by extension*, consisting in adding new fields or methods. A method to fix a new position of a point is introduced:

$$\begin{array}{l}\mathsf{resetablePoint} = \lambda x\,\mathsf{mixin}\,(\\ \qquad \mathsf{inherit}(\mathsf{point}\,x),\\ \qquad \mathsf{meth}\ \mathsf{reset}(z,s) = \lambda x(s.\mathsf{pos} := x)\\ \qquad\ )\end{array}$$

We can now give another example of inheritance by restriction, similar to the one of stacks from dequeues in Snyder (1986a): we may decide to restrict the use of the reset method to put the point back to its initial position, and to remove access to

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

this method:

$$\begin{array}{l}\mathsf{clearablePoint} = \lambda x\, \mathsf{mixin}\,(\\ \qquad\qquad \mathsf{inherit}(\mathsf{resetablePoint}\, x),\\ \qquad\qquad \mathsf{meth\ clear}(z,s) = (z \Leftarrow \mathsf{reset})x,\\ \qquad\qquad \mathsf{without\ reset}\\ \qquad\qquad\quad)\end{array}$$

A feature which we might wish to have is the ability to perform some given procedure at each object creation from a class. For instance, we may wish to have a counter incremented, or some warning printed each time an object is created. The simplest way to do that is to prefix the definition of the class with the appropriate procedure, as follows:

$$\begin{array}{l}\mathsf{let\ iPoint} = \mathsf{let}\ n = \mathsf{ref}\ 0\\ \qquad \mathsf{in\ let\ init} = \lambda x\,(n := !n + 1)\,;\mathsf{print\_string}\ \text{"new point number "};\\ \qquad\qquad\qquad \mathsf{print\_int}\ !n\,;\mathsf{print\_string}\ \text{" at "}\,;\mathsf{print\_int}\ x;\\ \qquad\qquad\qquad \mathsf{print\_newline}();\\ \qquad\quad \mathsf{in}\ \lambda x\,(\mathsf{init}\, x)\,;(\mathsf{point}\, x)\\ \mathsf{in}\ \cdots\end{array}$$

We could also incorporate the initialisation procedure as a method in the class, thus gaining the ability to invoke other methods. In this case, to create an object we would use the function:

$$\mathsf{newInit} = \lambda m(\mathsf{let}\ o = \mathsf{new}\, m\ \mathsf{in}\ o \Leftarrow \mathsf{init}\,;o\backslash\mathsf{init})$$

All the previously introduced classes are typable, and objects may be created from them. Up to now, the form of inheritance that we have used is quite standard in that we always specified the inherited superclass – some kind of point – while adding, modifying or removing some of its ingredients. However, mixins are really means to transform classes, getting new ones by "applying" the mixin to various superclasses, thus *reusing the transformation*. Here is a more mixin-oriented example: coloring an object. We define the coloring mixin as follows:

$$\begin{array}{l}\mathsf{let\ coloring} = \lambda c\, \mathsf{mixin}\,(\\ \qquad\qquad \mathsf{var\ color} = c,\\ \qquad\qquad \mathsf{meth\ paint}(z,s) = \lambda y(s.\mathsf{color} := y)\\ \qquad\qquad\quad)\\ \mathsf{in}\ \cdots\end{array}$$

Then, in the scope of this declaration, we can use *inheritance by composition* to get

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

a colored version of a variety of classes, for instance:

$$\begin{aligned}\mathsf{colorPoint} = \lambda x \lambda c\, \mathsf{mixin}\,(&\\ &\mathsf{inherit}(\mathsf{point}\, x),\\ &\mathsf{inherit}(\mathsf{coloring}\, c)\\ &)\end{aligned}$$

$$\begin{aligned}\mathsf{colorRectangle} = \lambda x \lambda y \lambda c\, \mathsf{mixin}\,(&\\ &\mathsf{inherit}(\mathsf{coloring}\, c),\\ &\mathsf{inherit}(\mathsf{rectangle}\, x\, y)\\ &)\end{aligned}$$

The colorPoint for instance is typable, because we may unify the types that the self parameter has in the point and coloring classes, which are, respectively, $\langle t_1, \mathsf{pos} : \mathsf{int}\,\mathsf{ref}\rangle$ and $\langle t_1', \mathsf{color} : t_0'\,\mathsf{ref}\rangle$. Notice that the polymorphism associated with the let-construct, and more specifically the polymorphism offered by Wand's row variables is crucial here for the inheritance mechanism to work properly, where one usually employs some form of subtyping (see Bruce *et al.* (1997) and Fischer & Mitchell (1995)). This kind of polymorphism allows us in particular to reuse the coloring mixin in various contexts. Obviously, the type system would reject a composition of mixins making incompatible requirements about the type of self, like for instance having a common field with different (i.e. non-unifiable) types. Although we do not use a subtype relation, it is obviously possible to write explicit coercion functions, like for instance:

$$\mathsf{colorless} = \lambda x(x \backslash \mathsf{color} \backslash \mathsf{paint})$$

Notice that $(\mathsf{colorless}\, x)$ is the "same" object as x, in the sense that they share the same state, apart from what regards the color. Alternatively, one can use pattern-matching (see Jategaonkar & Mitchell (1993)) to "extract a sub-object":

$$\mathsf{asPoint} = \lambda x \langle \mathsf{pos} = x.\mathsf{pos}, \mathsf{move} = x.\mathsf{move}\rangle$$

To get a less structural view of subtyping, we could also attach this function as a method of the point class, writing it as: $\mathsf{meth}\ \mathsf{asPoint}(z, s) = \langle \mathsf{pos} = s.\mathsf{pos}, \mathsf{move} = s.\mathsf{move}\rangle$. Again, $(\mathsf{asPoint}\, x)$ is, as a point, the "same" object as x. We cannot use a similar technique to make a copy of a point, since it is not possible to break the (recursive) binding of self in the move method. We should instead use another function, namely:

$$\mathsf{copyAsPoint} = \lambda x\, \mathsf{new}(\mathsf{point}(!x.\mathsf{pos}))$$

Since a mixin is a class transformer, it may introduce some new ingredients – fields or methods – while relying on the fact that some other ingredients will be provided by the superclass – these could be called "virtual". This shows up in the type of self (or super), and therefore the type system will reject any attempt to create an object instance of such an *abstract class*. An example is:

$$\mathsf{resetPos} \;=\; \mathsf{mixin}(\mathsf{meth}\ \mathsf{reset}(z, s) = \lambda x(s.\mathsf{pos} := x))$$

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

This mixin can be reused (inherited), but not instantiated, since typing (new resetPos) would involve solving the equation $\langle t_1, \mathsf{pos} : t_0\ \mathsf{ref}\rangle = \langle \mathsf{meth}\ \mathsf{reset} : t_0 \to \mathsf{unit}\rangle$, which is impossible. This mixin is intended to be used in combination with another one that introduces a pos field. To illustrate the use of the renaming facility, we elaborate on the last example: we redefine the reset method so that it updates not only the position, but also the color of an object (of some superclass), while keeping the possibility of updating the position only.

$$\begin{array}{l}\mathsf{resetPosColor} = \mathsf{mixin}\,(\\ \qquad\qquad \mathsf{rename\ reset\ as\ resetPos},\\ \qquad\qquad \mathsf{meth\ reset}(z,s) = \lambda x\lambda c(z \Leftarrow \mathsf{resetPos}\,x\ ;\ z \Leftarrow \mathsf{paint}\,c)\\ \qquad\qquad)\end{array}$$

Notice that it would be wrong – and the type system would complain – using overriding to define the reset method here, because after renaming it into resetPos, it is no longer present. The next kind of point illustrates a form of *multiple inheritance*:

$$\begin{array}{l}\mathsf{richPoint} = \lambda x\lambda c\,\mathsf{mixin}\,(\\ \qquad\qquad \mathsf{inherit}(\mathsf{resetablePoint}\,x),\\ \qquad\qquad \mathsf{inherit}(\mathsf{coloring}\,c),\\ \qquad\qquad \mathsf{inherit\ resetPosColor}\\ \qquad\qquad \cdots)\end{array}$$

We have omitted some parentheses here, and it should indeed be possible to prove that inheritance by composition is associative, with respect to some observational semantics. The order in which the components are introduced in the inheritance chain is also sometimes irrelevant: we could commute $\mathsf{inherit}(\mathsf{resetablePoint}\,x)$ and $\mathsf{inherit}(\mathsf{coloring}\,c)$ in the example without affecting the result, but the type system rejects inconsistencies – definition of methods (or fields) already present or required – that would arise if we had written one of these two ingredients after inheriting resetPosColor. Then multiple inheritance is restricted here to a *linear* pattern where the ingredients are introduced from left to right, and can be used or overriden subsequently, but not re-introduced. Typically, inheriting twice the same class, either directly or indirectly, is forbidden, if this class introduces some new fields or methods which are not removed or renamed.

An issue that we have not discussed is that of the *visibility* of the ingredients of a class. In our previous examples, one can always "externally" update the state of a point p, that is, its position, simply by executing a statement $p.\mathsf{pos} := \cdots$ However, in some cases we would like to forbid such a manipulation (see Fischer & Mitchell (1995) for some examples). In many object-oriented programming languages, the visibility of the fields, and of some of the methods of a class is, implicitly or explicitly, restricted to inheriting classes and/or to objects instances of the same class. In this paper, we do not investigate this issue, leaving this for further work, but we merely notice that one may always use the let construct to achieve state

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

encapsulation, as in

$$\begin{array}{l}\mathsf{secretPoint} = \lambda x\,\mathsf{let\ pos} = \mathsf{ref}\ x\\ \qquad\qquad\qquad\qquad \mathsf{in\ mixin}\,(\\ \qquad\qquad\qquad\qquad\quad \mathsf{meth\ move}(z,s) = \lambda d(\mathsf{pos} := \mathsf{!pos} + d))\\ \qquad\qquad\qquad\qquad\quad \mathsf{meth\ get}(z,s) = \mathsf{!pos}\\ \qquad\qquad\qquad\qquad\qquad)\end{array}$$

for instance. This approach, where the fields are strictly private to objects – but may be given access to by means of methods – is the one advocated by Snyder (1986a, 1986b). We could indeed have adopted a strict interpretation of fields (this can actually be done simply by removing the related constructs), but we think in some cases it may be worth having more opportunities for inheritance.

A feature we would like to add to our language is the ability to return, or send self. For instance, we may wish to attach the colorless function as a method of the coloring mixin, writing it as

$$\mathsf{meth\ colorless}(z,s) = s\backslash\mathsf{color}\backslash\mathsf{paint}$$

Another typical example is a method for *cloning* an object, that we could define([5]) using recursive classes, as in Cook *et al.* (1994):

$$\begin{array}{l}\mathsf{let\ rec\ clonablePoint} = \lambda x\,\mathsf{mixin}\,(\\ \qquad\qquad\qquad\qquad \mathsf{inherit(point}\,x),\\ \qquad\qquad\qquad\qquad \mathsf{meth\ clone}(z,s) = \mathsf{new(clonablePoint(!}s\mathsf{.pos))}\\ \qquad\qquad\qquad\qquad\quad)\end{array}$$

However, to be able to type this class, and instances of the coloring mixin extended with the colorless method, we must extend the type language with recursive record types $\mu t.\rho$. This is also necessary for typing a "subject/observer" interaction, where a subject notifies itself to an observer, as in:

$$\begin{array}{l}\mathsf{let\ subject}\ =\ \lambda o\,\mathsf{mixin\,(meth\ notify}(z,s) = \lambda y.y\ o\ s)\\ \mathsf{in\ let\ window}\ =\ \lambda x\lambda o\,\mathsf{mixin}\,(\\ \qquad\qquad\qquad\qquad \mathsf{inherit(point}\,x),\\ \qquad\qquad\qquad\qquad \mathsf{inherit(subject}\,o),\\ \qquad\qquad\qquad\qquad \mathsf{meth\ move}(z,s) \leftarrow \lambda d(z \Leftarrow \mathsf{move}\,d;\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad s \Leftarrow \mathsf{notify}(\lambda o.o \Leftarrow \mathsf{moved})),\\ \qquad\qquad\qquad\qquad \mathsf{meth\ draw}(z,s) = \cdots\\ \qquad\qquad\qquad\qquad\quad)\\ \quad \mathsf{in\ let\ manager}\ =\ \mathsf{mixin\,(meth\ moved}(z,s) = \lambda w.w \Leftarrow \mathsf{draw})\\ \qquad \mathsf{in\ let}\ m = \mathsf{new\ manager}\\ \qquad\quad \mathsf{in\ new\,(window}\,42\,m)\end{array}$$

[5] This is just an example. We do not claim that this is the right way to do clones.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

which, assuming that the code for the draw method is of type unit, has a type Window satisfying the equation:

$$\begin{aligned}\mathsf{Window} = \langle & \mathsf{pos} : \mathsf{int\ ref} \\ & \mathsf{meth\ move} : \mathsf{int} \to \mathsf{unit} \\ & \mathsf{meth\ notify} : \mathsf{Manager} \to \mathsf{Window} \to \mathsf{unit} \\ & \mathsf{meth\ draw} : \mathsf{unit} \rangle \\ \mathsf{Manager} = \langle & \mathsf{meth\ moved} : \mathsf{Window} \to \mathsf{unit} \rangle\end{aligned}$$

Notice that one could modify the notify method, so that it transmits a restricted view of the current object, like for instance:

$$\mathsf{meth\ notify}(z, s) \leftarrow \lambda y.z \Leftarrow \mathsf{notify}\ y\ o\ (s \backslash \mathsf{pos} \backslash \mathsf{move})$$

Sending objects, including self, to other objects is a natural way of programming in a higher-order, functional language, and therefore adding recursive record types is a natural extension to consider, which does not interfere with the typing of safe recursion. It would be interesting to see how our model, thus extended with recursive types, supports programming of various patterns that have emerged from object-oriented programming practice (cf. Gamma *et al.* (1994)).

7 Related work

The issue of imposing restrictions on recursive definitions to ensure that they define something is not at all a new one: many examples of notions of "contractive" or "guarded" recursive definitions may be found, for instance in formal language theory (Greibach's normal form of context-free grammars), process calculi or co-inductive types theories. In a language like ML, recursion is usually restricted to the form (let rec $f = \lambda x N$ in M), but as we discussed in the introduction, this does not suit our purpose. This has been recently generalized to deal with recursive modules by Crary *et al.* (1999), who use a "valuability" predicate, drawing upon Moggi's existence predicate. However, this again does not suit our purpose: for one thing, we wish to accept recursive expressions with evaluation that yields some computational effects. Moreover, to create objects instance of a class, we need to accept expressions like (let rec $x = (Gx)$ in M), as we have pointed out, and this – hence in particular our fixpoint combinator $\mathsf{fix} = \lambda f(\mathsf{let\ rec}\ x = fx\ \mathsf{in}\ x)$ – is rejected by the valuability system of Crary *et al.* (1999). As far as I can see, no obvious solution to this specific problem emerges from the literature.

The type system that I introduced in this paper has been extended and used to deal with recursive modules by Hirschowitz & Leroy (2002). Their extension is quite natural: considering that putting a variable underneath an abstraction increments its degree, whereas its degree is decremented when it occurs within a function applied to some argument, one may easily imagine that degrees could be integers, rather than booleans. One may even add ∞ as a degree, for a variable that does not occur free in an expression. This is what is done in Hirschowitz & Leroy (2002), and for instance (let rec $x = (\mathsf{F}x)\mathsf{F}$ in $\cdots$) is accepted in such a system. One probably even

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

could axiomatize the properties of an abstract notion of degree, generalizing what has been done here and in Hirschowitz & Leroy (2002). I have not followed this way here, because one of the main purposes was to generalize the type assignment algorithm of ML, without having to handle complex constraints on degrees. I recall that the only constraints to be solved here regarding degrees have the form $\alpha = 1$, which, when α is a constructed degree $\alpha_0 \wedge \alpha_1$, reduces to $\alpha_0 = 1$ and $\alpha_1 = 1$.

Regarding the modelling of objects, the literature is quite rich – we have mentioned a bit of it in the introduction. We did not formally compare our model to other ones, but it should be clear that, as far as objects (not mixins) are concerned, what we propose is very close to Reddy's denotational semantics of classes and objects (Reddy, 1988), and to the cyclic record semantics (see Abadi & Cardelli (1996), with the difference that we do not have to use assignment to model method override). Reddy did not deal with types however (nor with operational semantics), while this is our main concern. The mixin-based constructs we proposed are very close to the ones introduced by Bracha in the design of JIGSAW (Bracha, 1992); they are here integrated, by means of a formal interpretation, into an implicitly typed language. As we said in the introduction, most of the proposed models that include the main features of object-oriented programming use higher-order type theories, for which a type assignment algorithm is not available. Palsberg has explored the type inference problem for fragments of Abadi and Cardelli's object calculi (Palsberg, 1995; Palsberg & Jim, 1997), but these calculi do not have the principal type property. As a matter of fact, with the exception of OCAML[6], none of the object models we have cited supports principal type inference *à la* ML.

The OCAML's model (Rémy & Vouillon, 1998) integrates a class-based object layer into the ML language. The operational semantics is self-application, that is, the current object is substituted for self in the body of the method which is invoked. However, methods are not functions of self (but nonetheless regarded as values), as in Abadi and Cardelli's calculi for instance: self is a "special variable" that occurs free in the methods, as in the recursive record semantics of objects. As a consequence, typing is similar to the one of Wand's model, with specific type constructions for classes and objects, though class types are not arrow types, but depend upon the type of self given by the typing context. This is not exactly the model of the OCAML language (Leroy *et al*., 2000) however, in which a class body begins with a statement object(s), which is a binding for a self parameter. This is needed if one has to program with nested classes for instance. The semantics of this specific binder is not formally described; a class body is a value, but when an instance object is created, the scope of this binder is opened to evaluation (more precisely, the state part of the object is evaluated, but not the method part).

Apart from the fact that OCAML is, as a programming language, obviously much more elaborate than the preliminary design we proposed, there are some differences, and also strong similarities between the two models. Indeed, we do not claim our object model is by itself original: as we said, it is very close to the one of JIGSAW

[6] Also obviously Wand's model, which is however not expressive enough. The OML language of Reppy & Riecke (1996) also supports principal typing, but not inheritance.

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

(Bracha, 1992), and was also strongly inspired by the one of OCaml. For instance, although in OCaml classes are not first class, and there is no explicit "mixin" facility, the inheritance mechanism is very close to the one we have adopted – with a difference regarding how to use super, however. Also, we may use a restriction operation over classes, which is absent from OCaml. Another difference is that we do not require fields to be private to objects. In this paper, we have favoured an object encoding approach, rather than an object calculus approach – which is the one of OCaml – with the idea that this, in particular, should provide us with firm foundations for the typing of object-oriented constructs (cf. McQueen (2002)). One can see, for instance, that in OCaml, although the type system is very close to Wand's one, the typing of some constructions is not as general as it could be. For instance, method overriding is invariant, as far as types are concerned, and in the typing of classes, the type of self is required to extend the record type of methods, and this prohibits reusing classes by restricting their set of methods.

Encoding objects as recursive records obviously relies on some record calculus. One could in principle use standard, "fixed" records, as in Eifrig *et al.* (1995a) for instance, but this is not quite compatible with the idea of reusing code, since in inheriting from a class, one would have to explicitly include the method list of the superclass (as it is done also in Abadi & Cardelli (1996)). Various calculi of records that are "extensible" in some sense have been studied, starting with Wand's (1987) (some initial difficulties with principal typing were later solved (Jategaonakar & Mitchell, 1993; Ohori & Buneman, 1994; Rémy, 1994b; Wand, 1994)). For instance, various forms of record concatenation, symmetric or asymmetric, have been considered (Harper & Pierce, 1991; Rémy, 1994c; Wand, 1991). In this paper, we have chosen to use, mainly for a simplicity reason, a strict version of Cardelli & Mitchell's (1994) calculus, with a simpler typing however, where we express negative information by means of simple "annotations" rather than using subtyping and bounded quantification. Our record calculus is equivalent to the one of Jategaonkar & Mitchell (1993), who use a form of pattern-matching. There are some limitations with our choice: for instance, we only allow a limited form of (linear) multiple inheritance, but multiple inheritance is well-known to be difficult to manage (see Snyder (1986b), for instance). Also, one must be aware of the names of methods of the superclass when inheriting from it, in order to decide whether to use extension or over-riding. However, the type system warns the designer of the heir class about unintentional conflicts.

8 Conclusion

In this paper, we have adapted and extended Wand's typed model of classes and objects to an imperative setting, where the state of an object is a collection of mutable values. Our main achievement is the design of a type system which accepts safe let rec declarations of recursive values of any type (i.e. not just functional recursive values), while retaining the ability to construct a principal type for a typable term. The type assignment algorithm, as well as an interpreter of the language presented in this paper (including the mixin constructs, plus booleans, integers, recursive record types,

https://doi.org/10.1017/S0956796803004775 Published online by Cambridge University Press

etc.), have been implemented by Pascal Zimmer. The first experiments he has made confirm that, since the equations on degree expressions that we have to handle are very easy to solve, the task of building a principal type is not more complicated than in the standard case.

We believe that our type system does not impose any new restriction on the underlying language, where recursion is limited to (let rec $f = \lambda x N$ in M): it should not be difficult to show that a term of this language is typable, without using degrees, if and only if it is typable, with the "same" type, in our system, thus showing that our typing is a conservative extension of the usual one, if we forget about degrees. We also believe that our system can be extended to include more types, since only the core functional fragment is concerned with the technicalities arising from the degrees. Another issue to investigate is whether our approach may be applied to other situations where programming with recursive non-functional values could help – as I said above, this has already been done by Hirschowitz and Leroy regarding recursive modules. For instance, it should be easy to adapt our type system to call-by-value functional languages like SCHEME, as a static analysis for the let rec construct, keeping only the "degree" aspect, that is dealing with pseudo-types given by $\delta ::= \bullet \mid (\delta^a \to \delta)$. Another interesting topic is the question of how to implement the non-standard let rec construct considered here. Regarding this question, we refer to Boudol & Zimmer (2002), where an abstract machine is designed which implements correctly the functional core of our calculus.

Acknowledgements

The implementation done by Pascal Zimmer was of invaluable help to me. In particular, it allowed me to realise that previous versions of the type system were not powerful enough to accept the object-oriented constructs I had in mind, and to type-check the examples presented in the paper. I am also grateful to the referees for their comments and suggestions, which allowed me to improve the presentation of the paper, especially regarding the proofs in Section 4.

This work partially supported by the CTI "Objets Migrants: Modélisation et Vérification", France Télécom R&D, and by EU within the FET Global Computing initiative, project MIKADO IST-2001-32222.

References

Abadi, M. (1994) Baby Modula-3 and a theory of objects. *J. Functional Program.* **4**(2), 249–283.

Abadi, M. and Cardelli, L. (1996) *A Theory of Objects*. Springer-Verlag.

Ancona, D. and Zucca, E. (1998) A theory of mixin modules: basic and derived operators. *Math. Struct. Comput. Sci.* **8**, 401–446.

Bono, V., Patel, A., Shmatikov, V. and Mitchell, J. (1999a) A core calculus of classes and objects. *MFPS'99, Electronic Notes in Computer Science 20*. Springer-Verlag.

Bono, V., Patel, A., Shmatikov, V. and Mitchell, J. (1999b) A core calculus of classes and mixins. *ECOOP'99, Lecture Notes in Computer Science 1628*, pp. 43–66. Springer-Verlag.

Boudol, G. (2001) The recursive record semantics of objects revisited (extended abstract). *ESOP 2001, Lecture Notes in Computer Science 2028*, pp. 269–283. Springer-Verlag.

Boudol, G. and Zimmer, P. (2002) Recursion in the call-by-value lambda-calculus. Submitted.

Bracha, G. and Cook, W. (1990) Mixin-based inheritance. *ECOOP/OOPSLA'90*, pp. 303–311.

Bracha, G. (1992) *The Programming Language* JIGSAW*: Mixins, Modularity and Multiple Inheritance*. PhD Thesis, The University of Utah.

Bruce, K. (1993) Safe type checking in a statically-typed object-oriented programming language. *POPL'93*, pp. 285–298.

Bruce, K., Petersen, L. and Fiech, A. (1997) Subtyping is not a good "match" for object-oriented languages. *ECOOP'97: Lecture Notes in Computer Science 1241*, pp. 104–127. Springer-Verlag.

Cardelli, L. (1984) A semantics of multiple inheritance. *Semantics of Data Types: Lecture Notes in Computer Science 173*, pp. 51–67. (Also published in *Infor. & Computation*, **76** (1988).)

Cardelli, L. and Mitchell, J. (1994) Operations on records. In: Gunter, C. and Mitchell, M (editors), *Theoretical Aspects of Object-Oriented Programming*, pp. 295–350. MIT Press.

Cook, W. and Palsberg, J. (1989) A denotational semantics of inheritance and its correctness. *OOPSLA'89, ACM SIGPLAN Notices*, **24**(10), 433–443.

Cook, W., Hill, W. and Canning, P. (1994) Inheritance is not subtyping. In: Gunter, C. and Mitchell, M (editors), *Theoretical Aspects of Object-Oriented Programming*, pp. 497–517. MIT Press.

Crary, K., Harper, R. and Puri, S. (1999) What is a recursive module? *PLDI'99*, pp. 50–63.

Damas, L. and Milner, R. (1982) Principal type-schemes for functional programs. *POPL'82*, pp. 207–212.

Eifrig, J., Smith, S., Trifonov, V. and Zwarico, A. (1995a) An interpretation of typed OOP in a langage with state. *LISP & Symbolic Comput*. **8**, 357–397.

Eifrig, J., Smith, S. and Trifonov, V. (1995b) Sound polymorphic type inference for objects. *OOPSLA'95, ACM SIGPLAN Notices*, **30**(10), 169–184.

Fisher, K. (1996) *Types Systems for Object-Oriented Programming Languages*. PhD Thesis, Stanford University.

Fisher, K., Honsell, F. and Mitchell, J. (1993) A lambda calculus of objects and method specialization. *LICS'93*, pp. 26–38.

Fisher, K. and Mitchell, J. (1995) The development of type systems for object-oriented languages. *Theory & Practice of Object Systems*, **1**(3), 189–220.

Flatt, M., Krishnamurthi, S. and Felleisen, M. (1998) Classes and Mixins. *POPL'98*, pp. 171–183.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gunter, C. and Mitchell, J. (1994) *Theoretical Aspects of Object-Oriented Programming*. MIT Press.

Harper, R. and Pierce, B. (1991) A record calculus based on symmetric concatenation. *POPL'91*, pp. 131–142.

Hirschowitz, T. and Leroy, X. (2002) Mixin modules in a call-by-value setting. *ESOP'02: Lecture Notes in Computer Science 2305*, pp. 6–20. Springer-Verlag.

Jategaonkar, L. and Mitchell, J. (1993) Type inference with extended pattern matching and subtypes. *Fundamenta Informaticae*, **19**, 127–166.

Jouannaud, J.-P. and Kirchner, Cl. (1991) Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification. In: Lassez, J.-L. and Plotkin, G. (editors), *Computational Logic, Essays in Honor of A. Robinson*, pp. 257–321. MIT Press.

Kamin, S. (1988) Inheritance in SMALLTALK-80: a denotational definition. *POPL'88*, pp. 80–87.

Leroy, X., Doligez, D., Garrigue, J., Rémy, D. and Vouillon, J. (2000) The Objective Caml System, release 3.00. Documentation and user's manual, available `http://caml.inria.fr` (regularly updated).

Madsen, O. and Møller Pedersen, B. (1989) Virtual Classes: A powerful mechanism in object-oriented programming. *OOPSLA'89, ACM SIGPLAN Notices*, **24**(10), 397–406.

Martelli, A. and Montanari, U. (1982) An efficient unification algorithm. *ACM TOPLAS*, **4**(2), 258–282.

MacQueen, D. (2002) Should ML be object-oriented? *Formal Aspects Comput*. **13**(3–5), 214–232.

Meyer, B. (1986) Genericity versus inheritance. *OOPSLA'86, ACM SIGPLAN Notices*, **21**(11), 391–405.

Milner, R. (1978) A theory of type polymorphism in programming. *J. Comput. & System Sci.* **17**, 348–375.

Milner, R., Tofte, M., Harper, R. and MacQueen, D. (1997) *The Definition of Standard ML* (revised). MIT Press.

Ohori, A. and Buneman, P. (1994) Static type inference for parametric classes. In: Gunter, C. and Mitchell, M (editors), *Theoretical Aspects of Object-Oriented Programming*, pp. 121–147. MIT Press.

Palsberg, J. (1995) Efficient inference of object types. *Infor. & Computation*, **123**(2), 198–209.

Palsberg, J. and Jim, T. (1997) Type inference with simple selftypes is NP-complete. *Nordic J. Comput*. **4**(3), 259–286.

Pierce, B. and Turner, D. (1994) Simple type-theoretic foundations for object-oriented programming. *J. Functional Program*. **4**(2), 207–247.

Reddy, U. (1988) Objects as closures: abstract semantics of object-oriented languages. *ACM Symposium on LISP and Functional Programming*, pp. 289–297.

Rémy, D. (1994a) Programming with ML-ART: an extension to ML with abstract and record types. *TACS'94: Lecture Notes in Computer Science 789*, pp. 321–346. Springer-Verlag.

Rémy, D. (1994b) Type inference for records in a natural extension of ML. In: Gunter, C. and Mitchell, M. (editors), *Theoretical Aspects of Object-Oriented Programming*, pp. 67–95. MIT Press.

Rémy, R. (1994c) Typing record concatenation for free. In: Gunter, C. and Mitchell, M. (editors), *Theoretical Aspects of Object-Oriented Programming*, pp. 351–372. MIT Press.

Rémy, D. and Vouillon, J. (1998) Objective ML: an effective object-oriented extension of ML. *Theory & Practice of Objects Syst*. **4**(1), 27–50.

Reppy, J. and Riecke, J. (1996) Simple objects for Standard ML. *PLDI'96*, pp. 171–180.

Snyder, A. (1986a) CommonObjects: an overview. *ACM SIGPLAN Notices*, **21**(10), 19–28.

Snyder, A. (1986b) Encapsulation and inheritance in object-oriented programming languages. *OOPSLA'86, ACM SIGPLAN Notices*, **21**(11), 38–45.

Taivalsaari, A. (1996) On the notion of inheritance. *ACM Comput. Surv*. **28**(3), 438–479.

Wand, M. (1987) Complete type inference for simple objects. *LICS'87*, pp. 37–44.

Wand, W. (1991) Type inference for record concatenation and multiple inheritance. *Infor. & Computation*, **93**(1), 1–15.

Wand, M. (1994) Type inference for objects with instance variables and inheritance. In: Gunter, C. and Mitchell, M (editors), *Theoretical Aspects of Object-Oriented Programming*, pp. 97–120. MIT Press.

Wright, A. (1995) Simple imperative polymorphism. *LISP & Symbolic Comput*. **8**, 343–355.

Wright, A. and Felleisen, M. (1994) A syntactic approach to type soundness. *Infor. & Computation*, **115**(1), 38–94.